

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

OpenResty开源项目创始人力荐

Broadview[®]
www.broadview.com.cn

Nginx

罗剑锋 著

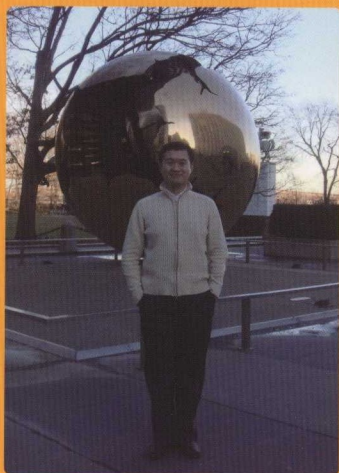
完全开发指南

使用C、C++和OpenResty

 中国工信出版集团

 电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

◀ 作者介绍 ▶



罗剑锋 (网名Chrono)

1996年就读于东北财经大学

1997年开始接触C/C++

1998年参加计算机软件专业技术资格和水平考试, 获高级程序员资质

2003年毕业于北京理工大学, 获计算机专业硕士学位

主要研究方向为C/C++、设计模式、高性能网络服务器开发

业余爱好是阅读、旅游、欣赏音乐和电影

Nginx

完全开发指南

使用C、C++和OpenResty

罗剑锋 著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

Nginx 是著名的 Web 服务器,性能优异,运行效率远超传统的 Apache、Tomcat,广泛应用于国内外诸多顶级互联网公司。

Nginx 的一个突出特点是其灵活优秀的模块化架构,可以在不修改核心的前提下增加任意功能,自 2004 年发布至今,已经拥有百余个官方及非官方的功能模块(如 proxy、mysql、redis、rtmp、lua 等),使得 Nginx 成长为了一个近乎“全能”的服务器软件。

Nginx 功能强大,架构复杂,学习、维护和开发的门槛较高。为了帮助读者跨越这一障碍,本书深入最新的 Nginx 源码(Stable 1.12.0),详细剖析了模块体系、动态插件、功能框架、进程模型、事件驱动、线程池、TCP/UDP/HTTP 处理等 Nginx 核心运行机制,在此基础上讲解如何使用 C、C++、Lua、nginScript 等语言来增强扩展 Nginx,让任何人都能够便捷、轻松地开发和定制 Nginx,进而应用到自己的实际工作中,创造出更多的价值。

本书结构严谨、脉络清晰、论述精确、详略得当、图文并茂,值得广大软件开发工程师、系统运维工程师和编程爱好者拥有。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Nginx 完全开发指南:使用 C、C++和 OpenResty / 罗剑锋著. —北京:电子工业出版社,2017.6
ISBN 978-7-121-31457-5

I. ①N… II. ①罗… III. ①互联网络—网络服务器—程序设计—指南②C 语言—程序设计—指南
IV. ①TP368.5-62②TP312.8-62

中国版本图书馆 CIP 数据核字(2017)第 094580 号

策划编辑:孙学瑛

责任编辑:安娜

印 刷:北京京科印刷有限公司

装 订:北京京科印刷有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编:100036

开 本:787×980 1/16 印张:39.75 字数:901 千字

版 次:2017 年 6 月第 1 版

印 次:2017 年 6 月第 1 次印刷

定 价:99.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819, faq@phei.com.cn。

前言

缘起

最早接触 Nginx 大概是在 2011 年，面对着一个全新的 Web 服务器，和大多数人一样最初我也是一片茫然，能找到的参考资料十分有限，安装、配置、运行几乎都是“摸着石头过河”，犯过许多低级错误。

随着对 Nginx 逐渐熟悉，它的高并发处理能力给我留下了深刻的印象，作为一个开源软件的爱好者，很自然地想要探究一下它的内部工作原理。我由此开始了对 Nginx 源码的钻研之路，中间经过了很多的艰辛曲折，走过不少的弯路。

我最常用的工作语言是 C++，所以在阅读 Nginx 源码时也总以 C++ 的面向对象方式来思考和理解，以对象作为切入点记笔记、画 UML：从最简单的 `ngx_str_t`、`ngx_array_t` 入手，然后到 `ngx_request_t`、`ngx_upstream_t` 等复杂的结构，再围绕着这些对象研究相关的功能函数和处理流程，梳理代码逻辑的同时也摸索着使用 C++ 编写 Nginx 模块的方法，逐渐积累了一些用起来颇为顺手的小工具——当然还是比较初级的形式。

三年多前，我被调到了新的工作岗位，需要重度使用 Nginx 开发，这让我以前的零散积累终于有了用武之地。那段时间里使用 C/C++ 陆续做了很多东西，也借着机会重新优化了原有的工具代码。

繁忙的工作之余，我有了种进一步整理经验的迫切感，因为只有系统完整地分享这些知识，才能让更多的人基于 Nginx 二次开发，让 Nginx 更好地为网络世界服务。

同一时间，市面上也出现了一些 Nginx 开发相关的资料、书籍，但在我看来却有“粗制滥造”之嫌：行文混乱，“车轱辘话”“口头禅”满天飞，甚至大段照抄指令说明，还有对源码

的曲解，未免有点儿“误人子弟”，读起来实在是难受。终于，在“忍无可忍”的心态之下，我动起了写作本书的念头。

经过近一年的努力，现在这本书终于呈现在了读者面前，结构上基本反映了我学习研究 Nginx 时的心路历程，从最初的“一无所知”起步，逐渐深入到定制开发的层次，希望能与读者“心有戚戚焉”。

Nginx 随感

毫无疑问，Nginx 是目前这个地球上所能获得的最强劲的 Web 服务器（没有之一），同时也是目前最成熟、最优秀的 TCP/HTTP 服务器开发框架。

Nginx 资源消耗低，并发处理性能高，配置灵活，能够连接 CGI、PHP、MySQL、Memcached 等多种后端，还有着出色的负载均衡能力，可以整合封装各种 service，构建稳定高效的服务。如今 Nginx 已经成为了网站架构里不可或缺的关键组件，广泛应用于国内外许多大型 IT 企业。每一个繁忙的网站背后，可能都有 Nginx 默默工作的身影。

在 Nginx 出现之前，使用 C/C++ 开发 Web 服务器是项比较“痛苦”的工作，虽然有很多网络程序库可以使用（例如 asio、libevent、thrift 等），但它们通常只关注较底层的基础功能实现，离成熟的“框架”相距甚远，不仅开发过程烦琐低效，而且程序员还必须要处理配置管理、进程间通信、协议解析等许多 Web 服务之外的其他事情，才能开发出一个较为完善的服务器程序。但即使开发出了这样的服务器，通常性能上也很难得到保证，会受到程序库和开发者水平等因素的限制——很长一段时间里，C/C++ 在 Web 服务器领域都没有大展拳脚的机会。

Nginx 的横空出世为 Web 服务器开辟了一个崭新的天地，它搭建了一个高性能的服务器开发框架，而且是一个完整的、全功能的服务器。模块化的架构设计很好地分离了底层支撑模块和上层逻辑模块，底层模块处理了配置、并发等服务器的外围功能，核心支撑模块定义了主体的 TCP/HTTP 处理框架。开发者可以把大部分精力集中在上层的业务功能实现上，再也不必去为其他杂事而分心，提高了软件的开发效率。

在 Nginx 框架里，C/C++ 程序员可以尽情发挥自己的专长，充分利用 Nginx 无阻塞处理的优势，打造出高质量的 Web 应用服务器，与其他系统一较高下。

Nginx 和 C/C++

Igor Sysoev 选择用 C 语言（准确地说是 ANSI C）来实现 Nginx 肯定是经过了认真

的考虑。

作为与 UNIX 一同诞生的编程语言，C 语言一直是系统级编程的首选。和其他高级语言相比，它简单可靠，更接近计算机底层硬件，运行效率更高。指针更是 C 语言的一大特色，善用指针能够完成许多其他语言无法完成的工作。

以 C 语言实现的 Nginx 没有“虚拟机”的成本，省略了不必要的中间环节，直接操纵计算机硬件，从根本上提高了 Web 服务器的处理能力。虽然 C 语言不直接支持面向对象，但 Nginx 灵活运用了指针，采用结构体+函数指针的形式，达到了同样的效果，从而使软件拥有了良好的结构。

C++是仅次于 C 的系统级编程语言，在兼容 C 的同时又增加了类、异常、模板等新特性，还支持面向对象、泛型、函数式、模板元等多种编程范式，可以说是计算机语言里的一个“庞然大物”。C++的特性很多，有的也很好用，但总体上的确比较复杂，易学难精，容易被误用和滥用，导致低效、难维护的代码，我想这可能是 Igor Sysoev 放弃使用 C++的一个重要原因。

另一个可能的原因是 C 语言本身已经非常稳定，几十年来没有太大的变动，在各个系统里都支持得非常好。而 C++在 1998 年才有了第一个标准，并且现在还在发展之中，语言特性还不够稳定（例如 `export`、`register` 等曾经的关键字在 C++11 里就已经被废弃），许多编译器对 C++的支持程度也有差异，这与 Nginx 的高可移植性目标明显不符。

但 C++毕竟还是有很多的优点，类可以更好地封装信息、异常简化了错误处理、模板能够在编译期执行类型计算。在 C++11 标准颁布之后，C++更是几乎变成了一门“全新”的语言，`auto`/`decltype`/`nullptr`/`noexcept` 等新关键字增强了语言的描述能力，标准库也扩充了相当多的组件，易用性和稳定性都大大提升。

在 Nginx 里使用 C++时要对 C++的长处和不足有清醒的认识，避免多层次继承、虚函数等影响效率的编程范式，只使用经过充分验证的、能够切实提高开发效率和性能的语言特性和库，避免华而不实的技术炫耀，尽量做到像 Nginx 源码那样质朴踏实。只有这样，才能够发挥出 $1+1>2$ 的作用，让 Nginx 从 C++中得到更进一步的发展动力。

Nginx 和 OpenResty

多年以前 Nginx 开发使用的语言只能是 C 和 C++，而现在，越来越多的开发者逐渐转向了 OpenResty，使 Lua 搭建高并发、高性能、高扩展性的 Web Server。

我接触 OpenResty 的时间并不算很长，大约在四年左右。由于 C/C++程序员“天生的傲

慢”，一开始对 OpenResty 确实有点儿“抵触情绪”，总觉得脚本程序比不上 C/C++ 实现。然而随着使用的增多，特别是在研究了它的源码之后，我不得不感慨 OpenResty 的精致、完美和强大，简直是所有 Nginx 开发者“梦寐以求的至宝”。

由于 agentzh 对 Nginx 的运行机制了如指掌，OpenResty 的核心部分——ngx_lua 一个模块就涵盖了 access/rewrite/content/log 等多个处理阶段，再搭配上小巧灵活的 Lua 和高效的 LuaJIT，我们就能够在更高级的业务层次上使用“胶水”代码来调用组合 Nginx 底层功能，轻松开发出丰富 Web 服务，极大地节约了宝贵的时间和精力。

当然，OpenResty 并不只有 ngx_lua，围绕着 ngx_lua 还有众多的库和辅助工具，构成了一个相当完善的生态环境，这些组件相互支撑，利用得当可以更好地提高生产效率。

OpenResty 现在正处于蓬勃发展的阶段，今后的 OpenResty 也许不仅限于 Nginx 和 Web Server，而将成为一个更通用的开发平台，工作语言也不仅限于 Lua，可能还会有其他新的语言（例如 agentzh 正在做的 edgelang 和 fanlang），让我们拭目以待。

致谢

首先当然要感谢 Nginx 的作者 Igor Sysoev，没有他就不会有如此优秀的 Web 服务器，也就不会有本书的诞生。

OpenResty 创始人章亦春（agentzh）是一位非常亲切随和的人，在 Nginx、DSL、Dynamic Tracing 等领域造诣极高，本书部分章节有幸经他审阅，在此表示最诚挚的谢意。

亲情永远是人生命中最值得珍惜的部分，我要感谢父母多年来的养育之恩和“后勤”工作，感谢妻子在生活中的陪伴，感谢两个可爱的女儿，愿你们能够永远幸福快乐。

最后，我也要感谢读者选择本书，希望读者能够在阅读过程中有所收获，在 Nginx 开发过程中获得乐趣。

您的朋友 罗剑锋

2017 年 4 月 28 日 于 北京 亚运村

目录

第 0 章 导读.....	1	1.3.2 进程配置.....	20
0.1 关于本书	1	1.3.3 动态模块配置.....	22
0.2 读者对象	2	1.3.4 运行日志配置.....	22
0.3 读者要求	3	1.3.5 events 配置	23
0.4 运行环境	4	1.3.6 http 配置	23
0.5 本书的结构	4	1.3.7 server 配置.....	25
0.6 如何阅读本书	7	1.3.8 location 配置.....	26
0.7 本书的源码	8	1.3.9 file 配置	27
第 1 章 Nginx 入门.....	9	1.3.10 upstream 配置.....	27
1.1 关于 Nginx	9	1.3.11 变量.....	28
1.1.1 历史.....	10	1.4 总结	30
1.1.2 特点.....	10	第 2 章 Nginx 开发准备	31
1.1.3 进程模型.....	11	2.1 开发环境	31
1.1.4 版本.....	12	2.1.1 C++标准	31
1.2 安装 Nginx	13	2.1.2 Boost 程序库	32
1.2.1 准备工作.....	13	2.2 目录结构	32
1.2.2 快速安装.....	14	2.3 源码特点	34
1.2.3 运行命令.....	14	2.3.1 代码风格.....	34
1.2.4 验证安装.....	16	2.3.2 代码优化.....	34
1.2.5 定制安装.....	16	2.3.3 面向对象思想.....	34
1.3 配置 Nginx	19	2.4 使用 C++.....	35
1.3.1 配置文件格式.....	19	2.4.1 实现原则.....	35

2.4.2 代码风格.....	36	3.6.4 日期操作函数.....	66
2.4.3 编译脚本.....	36	3.6.5 C++封装时间.....	67
2.5 C++包装类.....	38	3.6.6 C++封装日期.....	68
2.5.1 类定义.....	38	3.7 运行日志.....	70
2.5.2 构造和析构.....	39	3.7.1 结构定义.....	71
2.5.3 成员函数.....	40	3.7.2 操作函数.....	71
2.6 总结.....	40	3.7.3 C++封装.....	72
第3章 Nginx 基础设施.....	41	3.8 总结.....	74
3.1 头文件.....	41	第4章 Nginx 高级数据结构.....	77
3.1.1 Nginx 头文件.....	41	4.1 动态数组.....	77
3.1.2 C++头文件.....	42	4.1.1 结构定义.....	77
3.2 整数类型.....	42	4.1.2 操作函数.....	79
3.2.1 标准整数类型.....	43	4.1.3 C++动态数组.....	79
3.2.2 自定义整数类型.....	43	4.2 单向链表.....	83
3.2.3 无效值.....	44	4.2.1 结构定义.....	83
3.2.4 C++封装.....	44	4.2.2 操作函数.....	84
3.3 错误处理.....	47	4.2.3 C++迭代器.....	85
3.3.1 错误码定义.....	48	4.2.4 C++单向链表.....	87
3.3.2 C++异常.....	48	4.3 双端队列.....	90
3.4 内存池.....	50	4.3.1 结构定义.....	90
3.4.1 结构定义.....	51	4.3.2 操作函数.....	91
3.4.2 操作函数.....	51	4.3.3 C++节点.....	93
3.4.3 C++封装.....	52	4.3.4 C++迭代器.....	95
3.4.4 清理机制.....	54	4.3.5 C++双端队列.....	97
3.4.5 C++内存分配器.....	57	4.4 红黑树.....	101
3.5 字符串.....	58	4.4.1 节点结构定义.....	101
3.5.1 结构定义.....	59	4.4.2 树结构定义.....	102
3.5.2 操作函数.....	59	4.4.3 操作函数.....	103
3.5.3 C++封装.....	61	4.4.4 C++红黑树.....	104
3.6 时间与日期.....	64	4.5 缓冲区.....	108
3.6.1 时间结构定义.....	64	4.5.1 结构定义.....	108
3.6.2 时间操作函数.....	64	4.5.2 操作函数.....	110
3.6.3 日期结构定义.....	65	4.5.3 C++缓冲区.....	111

4.6 数据块链	113	第 6 章 Nginx 模块体系	139
4.6.1 结构定义	114	6.1 模块架构	139
4.6.2 操作函数	114	6.1.1 结构定义	139
4.6.3 C++节点	115	6.1.2 模块的签名	141
4.6.4 C++迭代器	117	6.1.3 模块的种类	142
4.6.5 C++数据块链	118	6.1.4 模块的函数指针表	143
4.7 键值对	120	6.1.5 模块的类图	144
4.7.1 简单键值对	120	6.1.6 模块的组织形式	145
4.7.2 散列表键值对	121	6.1.7 模块的初始化	147
4.8 总结	121	6.1.8 模块的动态加载	150
第 5 章 Nginx 开发综述	123	6.2 配置解析	152
5.1 最简单的模块	123	6.2.1 结构定义	152
5.1.1 模块设计	124	6.2.2 配置解析的基本流程	156
5.1.2 配置解析	124	6.2.3 配置数据的存储模型	157
5.1.3 处理函数	126	6.2.4 访问配置数据	163
5.1.4 模块集成	128	6.2.5 确定配置数据的位置	163
5.1.5 编译脚本和命令	129	6.2.6 配置解析函数	165
5.1.6 测试验证	130	6.2.7 配置数据的合并	166
5.2 开发基本流程	131	6.2.8 配置指令的类型	167
5.2.1 设计	131	6.3 源码分析	168
5.2.2 开发	132	6.3.1 ngx_core_module	168
5.2.3 编译	133	6.3.2 ngx_errlog_module	171
5.2.4 测试验证	133	6.4 C++封装	172
5.2.5 调优	133	6.4.1 ngx_module_config	172
5.2.6 流程图	133	6.4.2 ngx_module	176
5.3 编译脚本	134	6.4.3 ngx_take	178
5.3.1 运行机制	134	6.4.4 NGX_MODULE_NULL	180
5.3.2 使用的变量	135	6.5 C++开发模块	180
5.3.3 模块脚本	135	6.5.1 模块的基本组成	180
5.3.4 两种脚本格式	136	6.5.2 配置信息类	181
5.3.5 旧式编译脚本	136	6.5.3 业务逻辑类	183
5.4 总结	137	6.5.4 模块集成类	184
		6.5.5 实现源文件	186

6.5.6 增加更多功能	187	第 8 章 Nginx HTTP 请求处理	221
6.6 总结	187	8.1 状态码	221
第 7 章 Nginx HTTP 框架综述	191	8.2 请求结构体	222
7.1 框架简介	191	8.3 请求行	223
7.1.1 模块分类	191	8.3.1 请求方法	223
7.1.2 处理流程	192	8.3.2 协议版本号	224
7.1.3 请求结构体	194	8.3.3 资源标识符	224
7.1.4 请求的处理阶段	195	8.4 请求头	225
7.1.5 请求的环境数据	197	8.5 请求体	226
7.2 处理引擎	198	8.5.1 结构定义	226
7.2.1 函数原型	198	8.5.2 操作函数	227
7.2.2 处理函数的存储方式	198	8.6 响应头	227
7.2.3 内容处理函数	199	8.6.1 结构定义	228
7.2.4 引擎的数据结构	200	8.6.2 操作函数	228
7.2.5 引擎的初始化	201	8.7 响应体	229
7.2.6 引擎的运行机制	202	8.8 源码分析	229
7.2.7 日志阶段的处理	205	8.8.1 ngx_http_static_module	230
7.3 过滤引擎	205	8.8.2 ngx_http_not_modified_filter_	
7.3.1 函数原型	206	module	231
7.3.2 过滤函数链表	206	8.9 C++封装	232
7.3.3 过滤函数的顺序	207	8.9.1 ngxHeaders	232
7.3.4 过滤链表的运行机制	209	8.9.2 ngxRequestBody	235
7.3.5 请求体过滤	210	8.9.3 ngxRequest	236
7.4 源码分析	211	8.9.4 ngxResponse	238
7.4.1 ngx_http_static_module	211	8.10 开发 handler 模块	241
7.4.2 ngx_http_not_modified_filter_		8.10.1 模块设计	241
module	212	8.10.2 配置信息类	241
7.5 C++封装	213	8.10.3 业务逻辑类	242
7.5.1 ngxModuleCtx	213	8.10.4 模块集成类	243
7.5.2 ngxHttpCoreModule	215	8.10.5 实现源文件	245
7.5.3 ngxFilter	217	8.10.6 编译脚本	245
7.6 总结	219	8.10.7 测试验证	246
		8.11 开发 filter 模块	246

8.11.1 模块设计	246	9.5.2 ngxUpstreamHelper	283
8.11.2 配置信息类	246	9.5.3 ngxHttpUpstreamModule	285
8.11.3 环境数据类	247	9.5.4 ngxLoadBalance	287
8.11.4 业务逻辑类	248	9.6 开发 upstream 模块	288
8.11.5 模块集成类	251	9.6.1 模块设计	288
8.11.6 实现源文件	252	9.6.2 配置信息类	288
8.11.7 编译脚本	253	9.6.3 业务逻辑类	289
8.11.8 测试验证	253	9.6.4 模块集成类	292
8.12 总结	253	9.6.5 实现源文件	293
第 9 章 Nginx HTTP 请求转发	255	9.6.6 编译脚本	293
9.1 框架简介	255	9.6.7 测试验证	294
9.1.1 工作原理	256	9.7 开发 load-balance 模块	294
9.1.2 请求结构体	257	9.7.1 模块设计	294
9.1.3 上游结构体	258	9.7.2 配置信息类	294
9.1.4 上游配置参数	260	9.7.3 业务逻辑类	295
9.2 请求转发机制	261	9.7.4 模块集成类	297
9.2.1 回调函数	261	9.7.5 实现源文件	298
9.2.2 初始化	263	9.7.6 编译脚本	299
9.2.3 设置连接参数	264	9.7.7 测试验证	299
9.2.4 启动连接	265	9.8 总结	299
9.2.5 处理数据	265	第 10 章 Nginx HTTP 子请求	301
9.3 负载均衡机制	266	10.1 子请求简介	301
9.3.1 结构定义	267	10.1.1 工作原理	302
9.3.2 初始化模块入口	271	10.1.2 请求结构体	303
9.3.3 初始化地址列表	272	10.1.3 回调函数	304
9.3.4 初始化算法	274	10.1.4 待处理请求链表	306
9.3.5 执行算法	274	10.1.5 子请求存储结构	306
9.4 源码分析	275	10.2 子请求运行机制	307
9.4.1 ngx_http_memcached_module	275	10.2.1 创建子请求	307
9.4.2 ngx_http_upstream_ip_hash_		10.2.2 处理引擎	311
module	278	10.2.3 数据整理	312
9.5 C++封装	281	10.3 C++封装	314
9.5.1 ngxUpstream	281	10.3.1 ngxSubRequestHandler	314

10.3.2	NgxSubRequest.....	316	11.5.1	添加变量.....	341
10.4	数据回传模块.....	317	11.5.2	读写变量.....	343
10.4.1	模块设计.....	317	11.6	在模块里使用复杂变量.....	343
10.4.2	环境数据类.....	317	11.6.1	配置信息类.....	344
10.4.3	业务逻辑类.....	319	11.6.2	业务逻辑类.....	344
10.4.4	模块集成类.....	321	11.6.3	模块集成类.....	344
10.4.5	编译脚本.....	322	11.6.4	测试验证.....	344
10.5	在模块里使用子请求.....	323	11.7	总结.....	345
10.5.1	模块设计.....	323	第 12 章	Nginx 辅助设施.....	347
10.5.2	配置信息类.....	323	12.1	摘要算法.....	347
10.5.3	业务逻辑类.....	324	12.1.1	MD5.....	347
10.5.4	测试验证.....	327	12.1.2	SHA-1.....	348
10.6	总结.....	328	12.1.3	MurmurHash.....	349
第 11 章	Nginx 变量.....	329	12.1.4	C++封装.....	349
11.1	结构定义.....	329	12.2	编码和解码.....	352
11.1.1	变量值.....	329	12.2.1	CRC 校验.....	352
11.1.2	变量访问对象.....	330	12.2.2	Base64 编码解码.....	353
11.1.3	变量的存储.....	331	12.2.3	URI 编码解码.....	354
11.1.4	请求结构体.....	331	12.2.4	HTML 和 JSON 编码.....	355
11.2	运行机制.....	332	12.3	正则表达式.....	356
11.2.1	注册变量.....	333	12.4	共享内存.....	356
11.2.2	获取变量.....	333	12.4.1	结构定义.....	357
11.2.3	修改变量.....	334	12.4.2	操作函数.....	357
11.3	复杂变量.....	334	12.4.3	C++共享内存.....	358
11.3.1	结构定义.....	334	12.5	总结.....	359
11.3.2	运行机制.....	335	第 13 章	Nginx 进程机制.....	361
11.4	C++封装.....	335	13.1	基本系统调用.....	361
11.4.1	NgxVariableValue.....	336	13.1.1	errno.....	361
11.4.2	NgxVariable.....	337	13.1.2	getrlimit.....	362
11.4.3	NgxVarManager.....	339	13.2	进程系统调用.....	362
11.4.4	NgxVariables.....	340	13.2.1	getpid.....	362
11.4.5	NgxComplexValue.....	340	13.2.2	fork.....	363
11.5	在模块里使用变量.....	341			

13.2.3	waitpid.....	363	13.9.3	master 进程流程图.....	387
13.3	信号系统调用.....	364	13.9.4	worker 进程.....	388
13.3.1	kill.....	364	13.9.5	worker 进程流程图.....	389
13.3.2	sigaction.....	365	13.10	总结.....	390
13.3.3	sigsuspend.....	365	第 14 章	Nginx 事件机制.....	393
13.4	结构定义.....	365	14.1	基本系统调用.....	393
13.4.1	ngx_cycle_t.....	365	14.1.1	errno.....	394
13.4.2	ngx_core_conf_t.....	366	14.1.2	ioctl.....	394
13.4.3	ngx_process_t.....	367	14.1.3	setitimer.....	394
13.5	全局变量.....	368	14.1.4	gettimeofday.....	395
13.5.1	命令行相关.....	368	14.2	socket 系统调用.....	395
13.5.2	操作系统相关.....	369	14.2.1	socket.....	396
13.5.3	进程功能相关.....	369	14.2.2	bind.....	396
13.5.4	信号功能相关.....	370	14.2.3	listen.....	396
13.6	启动过程.....	370	14.2.4	accept.....	396
13.6.1	基本流程.....	370	14.2.5	connect.....	397
13.6.2	解析命令行.....	371	14.2.6	recv.....	397
13.6.3	版本和帮助信息.....	372	14.2.7	send.....	397
13.6.4	初始化 cycle.....	372	14.2.8	setsockopt.....	398
13.6.5	测试配置.....	374	14.2.9	close.....	398
13.6.6	发送信号.....	374	14.2.10	函数关系图.....	398
13.6.7	守护进程化.....	374	14.3	epoll 系统调用.....	399
13.6.8	启动工作进程.....	375	14.3.1	epoll_create.....	400
13.6.9	流程图.....	376	14.3.2	epoll_ctl.....	400
13.7	信号处理.....	377	14.3.3	epoll_wait.....	401
13.7.1	信号处理函数.....	377	14.3.4	LT 和 ET.....	401
13.7.2	发送信号.....	378	14.3.5	函数关系图.....	402
13.7.3	处理信号.....	378	14.4	结构定义.....	403
13.8	单进程模式.....	379	14.4.1	ngx_event_t.....	403
13.8.1	single 进程.....	379	14.4.2	ngx_connection_t.....	404
13.8.2	single 进程流程图.....	381	14.4.3	ngx_listening_t.....	405
13.9	多进程模式.....	382	14.4.4	ngx_cycle_t.....	407
13.9.1	产生子进程.....	382	14.4.5	ngx_os_io_t.....	408
13.9.2	master 进程.....	383			

14.4.6	ngx_event_actions_t.....	411
14.4.7	ngx_posted_events	413
14.4.8	结构关系图.....	415
14.5	定时器	415
14.5.1	红黑树.....	415
14.5.2	操作函数.....	416
14.5.3	超时处理.....	416
14.6	模块体系	419
14.6.1	函数指针表.....	419
14.6.2	模块的组织形式.....	420
14.6.3	核心配置.....	422
14.6.4	epoll 模块.....	423
14.7	全局变量	425
14.7.1	更新时间相关.....	425
14.7.2	事件机制相关.....	426
14.7.3	负载均衡相关.....	426
14.7.4	统计相关.....	427
14.8	运行机制	427
14.8.1	模块初始化.....	427
14.8.2	进程初始化.....	429
14.8.3	基本参数初始化.....	429
14.8.4	epoll 初始化.....	430
14.8.5	连接池初始化.....	431
14.8.6	监听端口初始化.....	433
14.8.7	初始化流程图.....	434
14.8.8	添加事件.....	435
14.8.9	删除事件.....	439
14.8.10	处理事件.....	440
14.8.11	接受连接.....	444
14.8.12	负载均衡.....	447
14.8.13	避免阻塞.....	452
14.9	总结	452

第 15 章 Nginx 多线程机制..... 455

15.1	eventfd 系统调用	455
15.2	pthread 系统调用	456
15.2.1	pthread_create.....	456
15.2.2	pthread_exit.....	457
15.3	结构定义	457
15.3.1	ngx_thread_task_t.....	457
15.3.2	ngx_thread_pool_queue_t.....	458
15.3.3	ngx_thread_pool_t.....	458
15.3.4	结构关系图.....	459
15.4	事件通知	460
15.4.1	函数接口.....	460
15.4.2	初始化	460
15.4.3	发送通知.....	461
15.4.4	处理通知.....	462
15.5	运行机制	463
15.5.1	完成任务队列.....	463
15.5.2	创建线程池.....	463
15.5.3	创建任务.....	464
15.5.4	投递任务.....	465
15.5.5	执行任务.....	466
15.5.6	任务完成回调.....	468
15.5.7	销毁线程池.....	468
15.6	在模块里使用多线程	469
15.6.1	模块设计.....	470
15.6.2	配置信息类.....	470
15.6.3	业务逻辑类.....	470
15.6.4	测试验证.....	474
15.7	总结	474

第 16 章 Nginx Stream 机制..... 477

16.1	模块体系	477
------	------------	-----

16.1.1	函数指针表	477
16.1.2	基础模块	478
16.1.3	核心模块	480
16.1.4	结构关系图	481
16.1.5	存储模型	482
16.2	监听端口	483
16.2.1	结构定义	483
16.2.2	解析配置	485
16.2.3	启动监听	489
16.3	处理引擎	491
16.3.1	阶段定义	491
16.3.2	函数原型	491
16.3.3	处理函数的存储方式	492
16.3.4	引擎数据结构	492
16.3.5	结构关系图	493
16.3.6	引擎的初始化	493
16.4	过滤引擎	495
16.4.1	函数原型	495
16.4.2	过滤函数链表	495
16.5	运行机制	496
16.5.1	会话结构体	496
16.5.2	创建会话	497
16.5.3	执行引擎	500
16.5.4	通用阶段处理	501
16.5.5	预读数据	502
16.5.6	产生响应数据	506
16.5.7	过滤数据	506
16.5.8	结束会话	506
16.6	开发 stream 模块	507
16.6.1	C++封装	507
16.6.2	discard 协议	508
16.6.3	time 协议	510
16.6.4	chargen 协议	512
16.7	总结	514

第 17 章 Nginx HTTP 机制 517

17.1	结构定义	517
17.1.1	ngx_http_state_e	517
17.1.2	ngx_http_connection_t	518
17.1.3	ngx_http_request_t	518
17.2	初始化连接	519
17.2.1	建立连接	520
17.2.2	等待数据	521
17.2.3	读取请求头	524
17.3	执行引擎	528
17.3.1	初始化引擎	528
17.3.2	通用阶段	530
17.3.3	改写阶段	531
17.3.4	访问控制阶段	532
17.3.5	内容产生阶段	533
17.4	处理请求体	534
17.4.1	丢弃缓冲区数据	535
17.4.2	读取并丢弃数据	536
17.4.3	读事件处理函数	537
17.4.4	启动丢弃处理	538
17.5	发送数据	540
17.5.1	发送初始化	540
17.5.2	事件处理函数	541
17.6	结束请求	543
17.6.1	释放请求资源	543
17.6.2	检查引用计数结束请求	544
17.6.3	检查状态结束请求	545
17.6.4	综合处理结束请求	546
17.7	总结	548

第 18 章 Nginx 与设计模式 551

18.1	设计模式简介	551
18.2	框架级别的模式	551
18.3	业务级别的模式	553

18.4 代码级别的模式	554	19.4.5 应用开发流程	584
18.5 总结	556	19.5 功能接口	585
第 19 章 OpenResty 开发	557	19.5.1 运行日志	585
19.1 简介	557	19.5.2 时间与日期	586
19.1.1 历史	558	19.5.3 变量	587
19.1.2 版本	559	19.5.4 正则表达式	587
19.1.3 组成	559	19.5.5 请求处理	588
19.1.4 性能	561	19.5.6 请求转发	590
19.1.5 安装	562	19.5.7 子请求	592
19.1.6 目录结构	563	19.5.8 定时器	592
19.1.7 命令行工具	564	19.5.9 共享内存	593
19.1.8 参考手册	565	19.6 应用实例	594
19.2 Lua 语言	566	19.6.1 处理请求	594
19.2.1 注释	566	19.6.2 过滤请求	595
19.2.2 数据类型	567	19.6.3 转发请求	596
19.2.3 变量	568	19.6.4 子请求	597
19.2.4 运算	569	19.7 Stream Lua 模块	598
19.2.5 语句	570	19.7.1 功能接口	598
19.2.6 函数	572	19.7.2 discard	599
19.2.7 表	574	19.7.3 time	599
19.2.8 标准库	575	19.7.4 chargen	600
19.2.9 模块	576	19.7.5 echo	600
19.3 LuaJIT	577	19.8 lua-resty 库	601
19.3.1 continue	578	19.8.1 core	601
19.3.2 bit	578	19.8.2 cJSON	602
19.3.3 ffi	579	19.8.3 redis	603
19.4 Lua 模块	581	19.9 总结	603
19.4.1 指令简介	581	第 20 章 结束语	605
19.4.2 配置指令	581	20.1 本书的遗憾	605
19.4.3 功能指令	582	20.2 下一步	605
19.4.4 指令关系图	584	20.3 临别赠言	606

附录 A 推荐书目	607
附录 B GDB 调试简介	609
附录 C Nginx C++模块简介	611
附录 D Nginx 的字符串格式化.....	613
附录 E nginxScript 简介	615

第 0 章

导读

0.1 关于本书

Nginx^①是由俄罗斯工程师 Igor Sysoev 开发的一个高性能 Web 服务器，应用于诸多顶级互联网公司，为全世界数以亿计的网民提供着出色的服务。根据某些权威公司分析统计，现在它已是市场份额第二大的 Web 服务器，并且仍然在快速增长，在 Top 1000 的网站中更是超过了 Apache 而荣登榜首。

除了最引人注目的高性能和高稳定性外，Nginx 的另一个突出特点是高扩展性，其灵活优秀的模块化架构允许在不修改核心的前提下增加任意功能。自 2004 年正式发布以来，Nginx 已经拥有了百余个官方及非官方的功能模块（如 proxy、rtmp、memcached、redis、mysql、lua 等），这使得 Nginx 超越了一般意义上的 Web 服务器，成为了一个近乎“全能”的应用服务器。

Nginx 功能强大，架构复杂，学习、维护和开发的门槛都很高，导致很多初学者“久仰大名”但却难以入门。本书将深入浅出地解析 Nginx 源码，详细讲解如何使用 C、C++ 和 Lua 语言来增强扩展 Nginx，并充分利用 C++ 和 Lua 的高级特性，让读者能够更加便捷、轻松地开发和定制 Nginx，应用到自己的实际工作中。

① 似乎很多人（包括 Nginx 官网）都会孜孜不倦地提醒 Nginx 的正确发音，所以作者在这里也不能“免俗”：Nginx 的正确发音应该是“engine eks”。不过作者（还有周围的许多同事）却更愿意像 UNIX/Linux 那样称它为“engine ks”——虽然这是一个“错误”的发音，但却简洁明快。

0.2 读者对象

本书适合以下各类读者：

- 基于 Nginx 进行二次开发的软件工程师；
- 基于 Openresty/Lua 进行二次开发的软件工程师；
- Nginx 运维工程师；
- Web 服务器开发者；
- 对 Nginx 架构、内部实现感兴趣的程序员；
- C/C++/Lua、计算机编程的爱好者和在校学生。

Nginx 采用进程池、事件驱动等方式来支持海量并发连接的处理，搭建了一个高性能高稳定性的服务器框架，在这个框架之内可以利用 Nginx 内部的各种构件编写模块，自由实现所需的业务功能。因此，国内外都有很多个人和企业以 Nginx 为平台进行二次开发，编写模块甚至直接定制 Nginx，进而提高网站的整体服务能力。本书由浅入深地详细介绍了 Nginx 模块开发的全过程，并且细致地分析了 Nginx 的内部运行机制，能够帮助软件工程师较快地熟悉 Nginx 原理，迅速地投入到实际开发工作中。

OpenResty 是由 OpenResty 社区（创始人 agentzh）维护的一个服务器开发包，核心组件是 Nginx 和 ngx_lua 模块，可以在 Nginx 框架内使用动态脚本语言 Lua 快捷实现业务逻辑，它既保持了 Nginx 高性能的优势，又没有 C 语言开发效率低的劣势，是目前服务器开发领域里的新生力量，前途不可限量。本书详细介绍了 OpenResty 开发相关的各个方面知识，即使是没有 C 语言基础的开发者也可以很快理解掌握，开发出高质量的网络服务程序。

Nginx 项目十分活跃，版本更新较快，新的功能不断增加，这使得市面上的相关书籍和网络资料常常会变得“过时”——只言片语、语焉不详或者答非所问^①，Nginx 运维工程师很可能会遇到运行、配置出现问题却无法解决的尴尬局面。然而 Nginx 是开源的，一切问题都可以在源码中找到答案。但十余万行的 Nginx 源码又有如汪洋大海，如果没有经验丰富的引航员指路，工程师难免会迷失方向，上下求索而不得。本书正是担当了这样的引航员角色，详尽地分析了 Nginx 的配置解析功能和 HTTP/TCP/UDP 处理流程，让运维工程师可以快速定位配置指令和它的实现逻辑，深层次地做好 Nginx 的运维工作。

对于那些自行开发 Web 服务器的程序员来说，Nginx 也是一个非常有意义的借鉴品。它采用 epoll/kqueue 等异步调用削减了系统成本，充分挖掘了计算机硬件的性能，让 CPU、

^① 即使是 Nginx 的官方 wiki 里也会有一些过时的资料和文档，常常误导用户。

磁盘、网卡等设备并发运行，比传统的多进程、多线程方式的服务器运行得更快，服务能力更强。本书深入剖析 Nginx 源码，解析了模块架构、进程模型、事件驱动、多线程等机制，学习借鉴这些大师级、教科书级的经典代码，无疑会让自己的开发功力更上一层楼，写出更好的软件产品。

虽然 Nginx 功能如此强大，但它的实现却非常清晰易读：文件组织良好，代码格式优美，内部隐藏的大量编程技巧、设计思想和架构更是值得仔细研究的无价之宝。对于每一位喜好编程、准备或者正在投身于软件/互联网行业的人来说，本书都是一个很好的起点，可以从零起步窥探到 Nginx 内部的真正精髓，学习到各种实用的技术和知识，增加自身的“含金量”。

本书还大量应用了 C++11 标准和 Boost 程序库，使用了 lambda 表达式、模板元编程等许多高级特性，可以看作是现代 C++ 编程范式的一次成功实践，读者可以接触到当前 C++ 最新、最前沿的技术。

0.3 读者要求

本书要求读者基本了解 C 语言和网络编程知识。

Nginx 本身是用 C 实现的，而本书主要使用 C++ 作为编程语言，所以读者应该具备足够的 C/C++ 知识，例如宏、指针、类型转换、封装、继承、异常等 C/C++ 特性，如果已有实际编程经验则更好。

在基本的 C++ 之上，本书还使用了 C++11 标准和 Boost 程序库来提高开发效率，要求读者对模板、泛型编程和 C++ 标准库有一定的了解，能够运用 vector、list 等常见的泛型容器和迭代器。

如果读者主要使用 OpenResty/Lua 开发，那么可以不必过多关心 C++，但基本的 C 语言知识还是必须的，可以更好地理解 Nginx/OpenResty 的工作原理。

Nginx 是一个 Web 服务器，所以熟悉网络编程对于学习本书是非常有益的。但由于 Nginx 封装了很多网络通信的底层细节，所以本书不要求读者对网络编程技术有很深刻的认识，但至少应该对 TCP 和 HTTP 通信协议有所了解。

此外，因为 Nginx 大多运行在 Linux 服务器上，读者还应该再学习一些 UNIX/Linux 和 Shell 编程的相关知识，才能够编写 Nginx 的编译脚本和运行维护。

附录 A 列出了一些技术书籍，涵盖了大部分本书要求掌握的 C/C++ 语言和网络编程技术知识，建议读者阅读本书时参考。

0.4 运行环境

Nginx 可以跨平台编译和运行, 支持 Linux、FreeBSD、macOS、Windows 等多种操作系统。但就目前市场来看, Linux 是应用最普及的服务器操作系统, 故本书的开发环境选用 Linux。

Linux 有很多的发行版本, 企业中使用较多的是偏重于稳定性的 CentOS 和偏重于易用性的 Ubuntu, 出于个人喜好的原因本书选择了后者 (请 CentOS 支持者见谅)。

下面是本书使用的具体环境:

- 操作系统 : Ubuntu 14.04.05 (Linux 4.4.0);
- Shell : 系统自带的 dash (注意不是 bash);
- 编译器 : 系统自带的 GCC 4.8.4 (支持 C++11 标准)。

0.5 本书的结构

Nginx 是一个非常庞大的系统, 内部结构错综复杂, 简单的循序渐进方式难以透彻地讲解清楚, 也不利于学习和掌握开发知识, 所以本书结合“知其所以然”再“知其然”的方式组织全书的章节: 首先介绍 Nginx 的基本知识作为入门, 然后由浅入深地解析 Nginx 的源码和架构, 理解了内部运行机制后再介绍如何开发定制 Nginx。

学习 Nginx 不研究其内部工作原理是不行的, 但解析得太深会导致源码太多, 文字过于晦涩难懂; 介绍得太浅又不能达到“知其所以然”的效果。所以本书只以中等深度研究 Nginx 源码和框架, 讲解关键的流程、原理和函数调用, 不涉及过于底层的实现细节。每个章节首先学习结构定义等静态模型, 然后配合图示研究工作原理、运行机制等动态模型, 最后结合 Nginx 官方源码, 通过实例讲解开发要点, 这种方式能够较好地覆盖 Nginx 开发的各个方面。

全书共 20 章, 大致可分为四部分:

第 1 章是入门知识, 简要介绍 Nginx 的安装和使用; 第 2 章至第 12 章重点讲解使用 C/C++ 语言开发定制 Nginx, 主要是“知其然”; 第 13 章至第 18 章则深入 Nginx 的内部, 剖析它的底层运行机制, 帮助读者深层次地理解 Nginx, 是“知其所以然”; 第 19 章专门介绍 OpenResty, 它可以用 Lua 语言快速开发出高性能的 Web 应用。

各章的内容简介如下。

- 第 1 章: Nginx 入门

简要介绍 Nginx 的历史和特点，以及如何编译、安装和配置 Nginx，可以当作是一本浓缩精华的 Nginx 使用手册。

■ 第 2 章：Nginx 开发准备

本章是开发 Nginx 模块前的准备工作，介绍本书使用的 C++11 标准和 Boost 程序库，还有 Nginx 源码的目录结构、基本的代码风格和特点。针对 Nginx 源码的特点提出了 C++ 的解决方案，实现了一个对 Nginx 数据结构的 C++ 封装类。

■ 第 3 章：Nginx 基础设施

剖析 Nginx 这样复杂的系统必须从最底层的基础设施开始，本章首先介绍了 Nginx 框架里基本的整数类型和错误码，然后再研究内存池、字符串、时间日期和运行日志，同时使用 C++11 标准进行了面向对象的封装，打造出方便易用的基础工具类。

■ 第 4 章：Nginx 高级数据结构

本章研究 `ngx_array_t`、`ngx_list_t`、`ngx_queue_t`、`ngx_buf_t`、`ngx_rbtrees_t` 等各种高级 Nginx 数据结构，它们类似 C++ 的标准容器，在 Nginx 框架里经常出现，必须很好地理解并掌握它们的特性和用法，C++ 封装能够让这些数据结构更容易使用。

■ 第 5 章：Nginx 开发综述

本章使用一个简单的例子介绍 Nginx 开发的基本流程，并讲解了如何编写模块集成脚本。

■ 第 6 章：Nginx 模块体系

本章详细剖析 Nginx 的模块架构和配置解析原理，这是 Nginx 开发必需的重要技术，也是理解 Nginx 架构的关键，最后实现了 C++ 封装类，开发出现代 C++ 风格的 Nginx 模块。

■ 第 7 章：Nginx HTTP 框架综述

本章深入研究 Nginx 的 HTTP 处理框架，解析 http 模块的核心工作原理和流程，包括处理阶段、处理引擎、过滤链表等重要概念。

■ 第 8 章：Nginx HTTP 请求处理

本章详细解析 `ngx_http_request_t` 结构，讲解请求头、请求体、响应头、响应体等数据成员和它们的操作方法，实现 Nginx 里最常用的 handler 模块和 filter 模块。

■ 第 9 章：Nginx HTTP 请求转发

本章解析 Nginx 的请求转发机制，它是 Nginx 反向代理的基础，实现转发请求的 upstream 模块和负载均衡的 load-balance 模块，访问外部的网络资源。

■ 第10章: Nginx HTTP子请求

本章讲解 Nginx 的子请求机制, 可以让 Nginx 像调用函数那样调用 location 里的功能, 发起多个子请求实现复杂的业务逻辑, 让 Nginx 变成一台强大的应用服务器。

■ 第11章: Nginx 变量

Nginx 内置非常灵活的变量机制, 是配置指令之外另一种与模块通信的方式, 增强了静态的配置文件与动态运行的模块之间的互操作性。本章剖析了变量机制的工作原理, 实现了易用的 C++封装类。

■ 第12章: Nginx 辅助设施

本章介绍 Nginx 提供的一些实用工具, 包括 MD5/SHA-1 摘要、CRC/Base64/URI 编解码、正则表达式、共享内存等, 它们是并发处理和协议解析之外实现一个完善的 Web 服务器所必需的辅助功能。

■ 第13章: Nginx 进程机制

Nginx 使用了进程池模式, 以一个 master 进程管理多个 worker 进程, 可以充分地利用多核 CPU。本章先介绍基本的 UNIX 系统调用, 然后配合图解详细研究 Nginx 的信号处理、启动过程、单进程和多进程的工作流程及要点。

■ 第14章: Nginx 事件机制

事件驱动机制是 Nginx 高性能的关键, 本章详细阐述 socket 系统调用、epoll 工作原理、事件模块体系和相关的数据结构、使用多幅流程图完整解析了连接池、监听端口、接受连接、处理网络和定时器事件、负载均衡等 Nginx 的核心功能。

■ 第15章: Nginx 多线程机制

Nginx 主要使用单线程加事件驱动, 但也对多线程提供了有限的支持, 本章介绍 Nginx 多线程机制的工作原理: 线程池+生产者/消费者, 还给出了一个实际的开发示例。

■ 第16章: Nginx Stream 机制

Nginx 从 1.9.0 开始引入了 Stream 框架, 支持直接处理 TCP/UDP 协议, 相当于是一个简化版的 HTTP 框架, 研究它可以更好地理解进程机制和事件机制在处理请求时的作用。本章解析 Stream 机制的模块体系和运行机制, 并实现了 discard、time 等基本的应用协议。

■ 第17章: Nginx HTTP 机制

本章是第7章的进阶内容, 结合事件机制, 使用源码辅以流程图, 详细介绍 HTTP 框架的关键运行机制, 包括初始化连接、阶段式处理引擎、读取数据、发送数据和结束请求等。

■ 第 18 章：Nginx 与设计模式

本章综合之前章节的内容，简要总结了在 Nginx 里应用的一些设计模式，帮助读者从设计模式的角度来进一步理解 Nginx 架构。

■ 第 19 章：OpenResty 开发

OpenResty 是一个“比 Nginx 更好的 Nginx”，它集成了众多优良的 Nginx 模块和库，可以用动态语言 Lua 快速开发出高质量高性能的 Web 应用。本章介绍了 OpenResty 相关的各方面知识，包括 Lua 语言、LuaJIT、配置指令、功能接口和开发示例，而且完全从零起步，编程初学者也可以较容易地掌握这些知识。

■ 第 20 章：结束语

本章给出了读者在阅读完本书后进一步学习研究 Nginx 的方向。

■ 附录

书末的附录是对本书正文的补充，列出了一些参考书籍，还有就是一些不宜放在正文里的比较琐碎的内容，比如 ngx_cpp_module 介绍、ngx_sprintf() 格式列表等。值得一提的是特别介绍了 nginxScript，可以使用 JavaScript 脚本灵活地定制 Nginx。

0.6 如何阅读本书

编程初学者或者初次接触 Nginx 的读者应当首先阅读第 1 章，了解什么是 Nginx，并在本书的指导下安装配置 Nginx，搭建自己的开发环境，然后学习后续章节。

对 Nginx 比较熟悉的读者可以跳过第 1 章，从第 2 章开始顺序阅读，以 Nginx 源码片段为出发点，学习如何使用 C/C++ 来开发 Nginx 模块。其中第 2 章至第 4 章是 C/C++ 开发的基础，第 5 章才能开始正式编写 Nginx 模块。

如果读者已经是一个比较熟练的 Nginx 开发者，那么可以直接阅读第 5、6 章及之后的章节，钻研 Nginx 的内部架构和实现原理，深入理解 Nginx 的运行机制。不过第 2 章至第 4 章也并非可以完全忽略，其中的 C++ 泛型、元编程等崭新的编程范式也许会让你眼前一亮，能够从一个全新的角度来审视 Nginx。

使用纯 C 语言开发的读者可以忽略书中标题含有“C++”的小节，而只阅读与 Nginx C 接口、源码相关的部分。

如果读者只想使用 OpenResty/Lua 开发 Nginx 应用，建议先阅读第 1 章，了解 Nginx 安装部署的基本知识，然后跳过之后的 Nginx 框架源码解析，直接阅读第 19 章，快速开始

OpenResty 的学习。不过还是希望读者有了一些实际经验后再回头研究前面的章节，只有深入理解 Nginx 内部细节，才能更好地开发 OpenResty 应用。

0.7 本书的源码

为方便读者利用本书学习研究 Nginx 和 OpenResty，作者在 GitHub 网站上发布了本书内所有 C/C++、OpenResty 示例程序和 Shell 脚本的源代码，地址是：

https://github.com/chronolaw/ngx_cpp_dev.git	#C++源码和开发示例
https://github.com/chronolaw/openresty_dev.git	#OpenResty 开发示例

对于想深入钻研的读者，还有另外两个 GitHub 项目可供进一步参考：

https://github.com/chronolaw/annotated_nginx.git	#Nginx 源码详细注解
https://github.com/chronolaw/favorite-nginx.git	#各种有用的 Nginx 资源

第 1 章

Nginx入门

Nginx 是一个高性能、高稳定性和高扩展性的轻量级 TCP、UDP、HTTP、反向代理和邮件代理服务器。它运行效率高，系统消耗低，使用操作系统提供的异步 I/O 调用可以无阻塞地处理上万的并发请求，是当今众多 Web 服务器中的佼佼者，被 Airbnb、Dropbox、GitHub、Hulu、WordPress 等许多知名网站所采用。

本章简要介绍 Nginx 的历史和特点，讲解如何在 Linux 上安装和配置 Nginx，帮助读者尽快熟悉 Nginx。

1.1 关于 Nginx

自 20 世纪 90 年代以来，Internet 和 World Wide Web 逐渐在全世界普及。早期并没有 Web 服务器的概念，Internet 上的资源只是一些简单的静态文件。慢慢地，Web 服务器出现了，它在用户和文件之间加入了一个中间层，响应用户的 HTTP 请求，从本地或者其他地方获取对应的资源，再返回给用户。

1995 年，著名的 Apache 诞生了。最开始它只是一个 NCSA HTTPd 服务器的修改版，但许多志愿者不停地为它添加补丁新增功能，使它迅速成为了那个年代最流行的 Web 服务器，甚至直到二十多年后的今天也是如此。

由于时代的限制，Apache 被设计为“重量级”的服务器，使用 fork 进程的方式响应 HTTP 请求。虽然近年来 Apache 增加了多线程、多核心等新特性，但基本架构无法改变，在动辄成千上万并发的情况下显得有些力不从心，性能严重下降。

本世纪初,随着 C10K 问题的提出^①,计算机界开始认真地研究 Web 服务器的开发模型。Linux、FreeBSD 等操作系统引入了 epoll、kqueue 等高效异步 I/O 接口,奠定了解决海量并发的 C10K 甚至 C100K 问题的基础,最终导致 Nginx 这个 Web 服务器新秀的出现。

1.1.1 历史

2002 年,在仔细研究了已有的 Web 服务器之后,当时任职于俄罗斯门户网站 Rambler 的工程师 Igor Sysoev 开始编写一个新的 Web 服务器,目标是解决高并发的 C10K 问题,并设计了完全不同于以往服务器的全新架构——这就是 Nginx。

2004 年,Nginx 正式开源,版本号为 0.1.0,由于其优异的性能、绝佳的稳定性和良好的扩展性等鲜明特点迅速吸引了大量关注。在开源大旗的鼓舞之下,无数开发者加入贡献行列,为它修复错误、编写功能模块,Nginx 从此踏上了迅猛发展的道路,从当初默默无闻的小卒成长为如今 Web 服务器界的巨人。

2011 年,Nginx 之父 Igor Sysoev 在旧金山正式成立公司,为 Nginx 提供商业服务,并发布了商业版本的 Nginx Plus。

1.1.2 特点

作为 Web 服务器的后起之秀,Nginx 能够战胜 Apache、Lighttpd、Jetty、Tomcat 等众多对手,获得顶级互联网公司的青睐,必然有它的独到之处。以开发者的视角来看,Nginx 的特点是:

■ 高性能

卓越的性能是 Nginx 最突出的特点。Nginx 完全使用 C 语言编写,采用事件驱动模型,并且有针对性地对操作系统进行了特别优化,可以无阻塞地处理海量并发连接,经过仔细调整配置参数还能够进一步释放潜力,帮助网站应对日益增长的巨大访问压力。

■ 高稳定性

高度的稳定性是 Nginx 的又一大特点。内存池避免了 C 程序常见的资源泄漏问题,模块化的架构使得各个功能模块完全解耦,消除了相互间可能造成的不良影响,而独特的 one master/multi workers 进程池设计则实现了自我监控和管理,保证即使 worker 进程发生严重错误也可以快速恢复。在实际应用中,Nginx 服务器一经启动,就可以稳定地运行数天甚至数月之久。

^① 参见 <http://www.kegel.com/c10k.html>。

■ 低资源消耗

Nginx 的代码质量极高，不使用传统的进程或线程服务器模型，没有进程或线程切换时的成本，而且它还使用了很多节约系统资源的编程技巧，例如使用 `accept4` 来减少内核调用的次数、使用 `writev` 集中发送数据、使用字符串引用而不是拷贝，可以把宝贵的 CPU 和内存资源更多地用于对外提供服务，提高并发支持能力。

■ 高扩展性

Nginx 的模块化架构是一个非凡的设计。Nginx 本身就是由各种位于不同层次的功能模块组合而成的，它也允许/鼓励任何人依据架构规范开发任意功能的模块——同时充分利用 Nginx 框架提供的各种高效机制，然后再完美地融合到 Nginx 之中。

基于高扩展性，Nginx 以模块的形式实现了丰富的功能，例如 `access`、`flv`、`gzip`、`proxy`、`ssl` 等，而广大开发者也编写了大量的第三方模块，实现了更多更有用的功能。这些模块都遵循一致的开发原则，可以在配置文件里灵活配置，让 Nginx 提供更好的网络服务。^①

以上的三高一低四个方面是从开发者的角度总结的 Nginx 的主要特点。当然 Nginx 的优势远不止这些，它能够运行在多种操作系统上，安装和配置都很容易，支持定制日志、平滑升级、策略限速、热部署等许多重要的运维功能。

高性能是品质，高稳定性是保障，低资源消耗是基石，而高扩展性则是 Nginx 生命力的源泉，使 Nginx 拥有了近乎“无限”的能力——如果现有的 Nginx 模块不能满足我们的需求，最佳的解决方案无疑是开发自己的模块，集成到 Nginx 里，在享受 Nginx 高性能、高稳定性的同时实现自己的业务逻辑。这正是作者撰写本书的目的。

1.1.3 进程模型

Nginx 采用了独特的 `one master/multi workers` 进程池机制，它是 Nginx 能够稳定运行、灵活配置的保证。

通常情况下，Nginx 会启动一个 `master` 进程和多个 `worker` 进程对外提供服务^②。`master` 进程又称监控进程，它并不处理具体的 TCP/HTTP 请求，只接收 UNIX 信号，管理和监控 `worker` 进程，所以工作比较“清闲”。`worker` 进程平等地竞争 `accept` 客户端的连接，执行 Nginx 主要的业务逻辑，使用 `epoll`、`kqueue` 等机制高效率地处理 TCP/HTTP 请求。

① 详细的功能模块资料可以参看官网 Wiki 的列表，这里不再一一列举。

② 如果开启了缓存功能，可能还会有 `cache` 进程，本书暂不讨论。

master 进程和 worker 进程使用操作系统提供的进程间通信机制——如信号、UNIX 域套接字、共享内存等互相通信，彼此独立又保持一定的联系，形成一个有机的整体。当某个 worker 进程发生错误意外中止时，master 进程会快速重新 fork 出新的 worker 进程，保持进程池的稳定（具体工作原理可参见第 13 章）。

Nginx 的进程模型如图 1-1 所示。

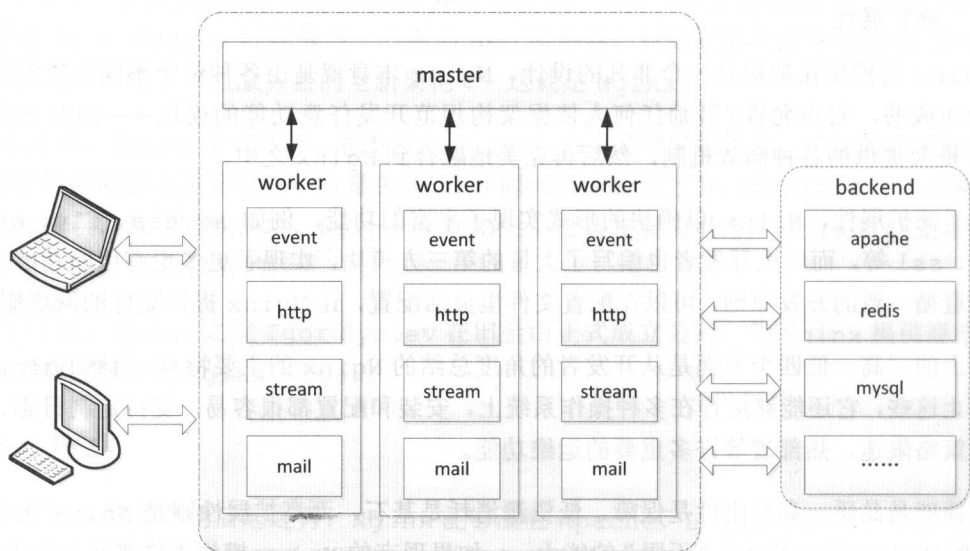


图 1-1 Nginx 的进程模型

在多核心的服务器上，Nginx 可以配置为每个进程运行在单独的核心，最大限度地减少 CPU 进程切换的成本，提高系统运行效率。

Nginx 也可以配置为不使用 master 进程，只有一个 worker 进程提供服务，这种方式不适用于真正的生产环境，但对于开发测试却很有用。

1.1.4 版本

本书不讨论商业版本的 Nginx Plus^①，只讨论开源版本的 Nginx，它的官网是：<http://www.nginx.org/>。

Nginx 官网提供三种类型的版本供用户选择：

^① 实际上商业版本的 Nginx Plus 是基于开源版 Nginx 的，同时添加了很多对于商业网站非常有用的实用功能，例如负载均衡、健康度检查、状态监控等。

- Mainline : 主线开发版本, 版本号为奇数。更新速度很快, 差不多每个月都会发布新版本, 汇集了最新的功能和错误修复, 但稳定性可能略差。
- Stable : 当前的稳定版本, 版本号为偶数。由 Mainline 版本 fork 而来, 经过了完全测试, 代码有一年的“冻结”期, 除非有重要 Bug 修复否则不会变动, 建议在正式生产环境中使用。
- Legacy : 历年曾经发布的稳定版本, 有特殊需求的话可以采用。

这三个版本的关系可以用图 1-2 来表示。

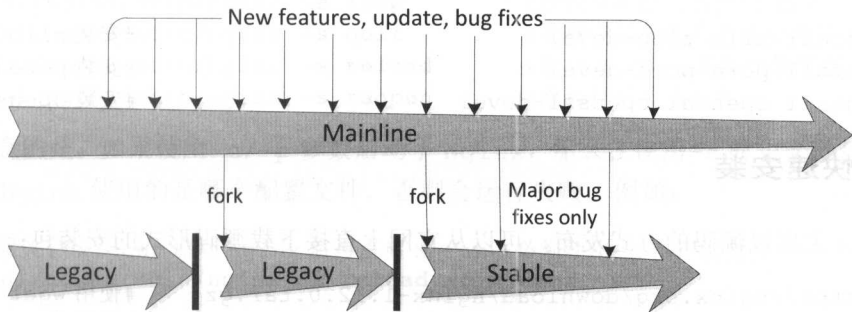


图 1-2 Mainline、Stable、Legacy 三个版本的关系

本书采用的 Nginx 是 2017 年 4 月发布的 Stable 1.12.0 版, 由于 Nginx 的核心架构和代码一直都比较稳定 (但局部实现偶尔会变动), 所以书中的讨论也基本适用于较早期的各种版本, 对于部分重要的版本差异会加以特别注解。

1.2 安装 Nginx

本节介绍在 Linux 操作系统上使用源码包的方式安装 Nginx, 但并不是一个完全的安装部署手册, 仅供读者参考。^①

1.2.1 准备工作

只要系统里有 GCC (或其他 C 编译器), Nginx 就可以编译安装, 但几个基本且重要的功能依赖于第三方库:

- zlib : 实现 gzip 压缩解压缩功能。

^① 有的 Linux 发行版也提供编译好的 Nginx 包, 可以使用 apt-get 或者 yum 安装, 但这种方式不够灵活, 不能定制模块, 本书不推荐。

- pcre : 实现配置文件里的正则表达式解析功能。
- openssl : 实现 SSL 功能。

在 Ubuntu 里安装这些依赖库的命令是:

```
sudo apt-get install libz-dev          #安装 zlib 库
sudo apt-get install libpcre3-dev      #安装 pcre 库
sudo apt-get install libssl-dev       #安装 openssl 库
```

使用其他操作系统的读者请参考网络相关资料, 例如对于 CentOS, 安装的命令可以是:

```
yum install zlib zlib-devel          #安装 zlib 库
yum install pcre pcre-devel          #安装 pcre 库
yum install openssl openssl-devel    #安装 openssl 库
```

1.2.2 快速安装

Nginx 主要以源码的方式发布, 可以从官网上直接下载源码形式的安装包:

```
wget http://nginx.org/download/nginx-1.12.0.tar.gz  #使用 wget 下载
tar xvfz nginx-1.12.0.tar.gz                        #解压缩
```

本书假定把 Nginx 解压缩到~/nginx 目录。

在~/nginx 目录下执行下面的命令即可快速安装 Nginx:

```
./configure      #编译前的配置工作
make             #编译
sudo make install #安装
```

这是最简单的 Nginx 安装方式, 没有任何的定制参数。Nginx 将会安装到默认的“/usr/local/nginx/”目录, 可执行文件是“/usr/local/nginx/sbin/nginx”, 默认配置文件是“/usr/local/nginx/conf/nginx.conf”。

1.2.3 运行命令

启动和停止 Nginx 需要以 root 身份, 或者使用 sudo。

下面介绍一些常用但不是全部的 Nginx 命令, 更详细的命令说明可以使用-h/-?参数查看, 或者查阅网络资料。

不带参数简单地执行程序就可以启动 Nginx 服务, 这将使用默认的配置文件的:

```
/usr/local/nginx/sbin/nginx          #使用默认的配置文件的启动 Nginx
```

我们也可以使用 `-c` 参数指定配置文件，这种方式允许启动多个不同的 Nginx 实例：

```
/usr/local/nginx/sbin/nginx -c x.conf      #指定配置文件 x.conf 启动 Nginx
```

`-p path` 是 `-c` 的增强版，用来设置工作目录，可以指定完整的 Nginx 环境：

```
/usr/local/nginx/sbin/nginx -p /opt/nginx  #设置 Nginx 的工作目录为 /opt/nginx
```

`-s signal` 参数可以快速地停止或者重启 Nginx，`signal` 值可以是 `stop`、`quit`、`reload` 或 `reopen`：

```
/usr/local/nginx/sbin/nginx -s stop      #强制立即停止 Nginx 服务
/usr/local/nginx/sbin/nginx -s quit      #处理完当前所有连接后再停止 Nginx
/usr/local/nginx/sbin/nginx -s reload    #重启 Nginx，重新加载配置文件
/usr/local/nginx/sbin/nginx -s reopen    #重新打开日志文件
```

需要注意的是，如果使用 `-c/-p` 参数启动了 Nginx，那么在使用 `-s` 时也必须使用 `-c/-p` 参数，告诉 Nginx 使用的是哪个配置文件，否则会运行失败。例如：

```
#在使用配置文件 x.conf 启动 Nginx 后再重启 Nginx，必须使用 -c 指定配置文件
/usr/local/nginx/sbin/nginx -s reload -c x.conf
```

```
#在使用 -p 启动 Nginx 后停止 Nginx，仍然要使用 -p 参数
/usr/local/nginx/sbin/nginx -s stop -p /opt/nginx
```

`-t` 或 `-T` 参数可以测试配置文件是否正确，后者同时还会打印出文件内容方便检查：

```
/usr/local/nginx/sbin/nginx -t      #检查默认的配置文
/usr/local/nginx/sbin/nginx -T      #检查默认的配置文并打印输出
/usr/local/nginx/sbin/nginx -t -c x.conf  #检查指定的配置文件 x.conf
```

`-v` 或 `-V` 参数可以显示 Nginx 的版本信息，它不需要 `root` 权限，两者的区别是 `-v` 可以显示更多的信息，包括 GCC 版本、操作系统版本、`configure` 参数等定制信息：

```
/usr/local/nginx/sbin/nginx -v      #显示简要的版本信息
nginx version: nginx/1.12.0          #Nginx 版本是 1.12.0
```

```
/usr/local/nginx/sbin/nginx -V      #显示完全的版本信息
nginx version: nginx/1.12.0          #Nginx 版本是 1.12.0
built by gcc 4.8.4 (Ubuntu 4.8.4)    #使用 Ubuntu GCC 4.8.4 编译
configure arguments:                  #没有特别的定制编译项
```

`-v` 参数很多时候特别有用，由于它给出了编译 Nginx 时的全部相关信息，所以我们可以使用其他版本的源码（例如升级或者修复 Bug）加上这些参数重新编译出一个功能完全相同的新程序。

1.2.4 验证安装

如果已经成功启动了 Nginx 服务,那么就可以使用 `wget` 或者 `curl` 这样的工具来验证 Nginx 是否正常工作。

Nginx 的默认配置文件开启了 `localhost:80` 服务,在 `/usr/local/nginx/html` 下存放有一些示例静态 html 文件, `curl` 测试命令是:

```
curl -vo /dev/null http://localhost/index.html      #curl 命令发送 HTTP 请求
```

如果 Nginx 正在运行,那么 `curl` 的部分输出可能如下:

```
* Connected to localhost (127.0.0.1) port 80 (#0)  #连接到 localhost:80

> GET /index.html HTTP/1.1                        #获取文件 index.html
> User-Agent: curl/7.35.0                          #curl 的版本号

< HTTP/1.1 200 OK                                  #响应码 200, 正常
< Server: nginx/1.12.0                             #服务器是 Nginx 1.12.0
< Content-Type: text/html                          #响应内容是普通文本
< Content-Length: 612                              #HTTP 正文长度是 612 字节
```

使用 Linux 的 `ps` 命令配合 `grep` 可以看到所有 Nginx 进程,也可以验证 Nginx 是否正常运行:

```
ps aux|grep nginx                                #ps 查看进程
root      8261    ... nginx: master process /usr/local/nginx/sbin/nginx
nobody    8262    ... nginx: worker process
```

从 `ps` 的输出我们可以看到当前共有两个 Nginx 进程,其中进程号为 8261 的是 master 进程,而 8262 号进程则是 worker 进程。

如果 Nginx 没有正确运行,我们可以查看它的错误日志以排除故障,默认的位置是“`/usr/local/nginx/logs/error.log`”。

1.2.5 定制安装

`make` 前执行的 `configure` 是 Nginx 的重要组成部件,它检查各种系统参数、命令行参数和依赖库,根据这些参数生成定制的 `Makefile` 和一些 C 源码文件,没有它就无法正确编译安装 Nginx。

虽然 `configure` 只是一个标准的 Shell 脚本,但其内部逻辑十分复杂,为了支持各种

操作系统、编译器和 CPU 做了大量工作，复杂程度甚至不逊于 Nginx 自身的 C 程序。^①

由于 configure 的选项很多，本书不能也没有必要完全罗列，仅列出一些个人认为比较重要的选项，其他可参考 help 或者网络资源。

--prefix=PATH:

配置 Nginx 安装部署的根目录，也就是工作目录。默认值是 “/usr/local/nginx/”，可以把它改为其他路径，这样就可以在一个系统里安装多个不同用途的 Nginx^②，类似的选项还有 --sbin-path、--conf-path 等。例如：

```
./configure --prefix=/opt/nginx #把 Nginx 安装到/opt/nginx
```

--with-stream:

启用 Nginx 的 stream 模块，让 Nginx 能够直接处理 TCP/UDP 协议。

--with-threads:

启用 Nginx 的线程池机制，允许 Nginx 使用多线程来处理数据。

--with-pcre=DIR, --with-openssl=DIR:

虽然 Linux 系统通常都内置 PCRE、OpenSSL 库，但有的时候可能版本比较低，功能不完善（更严重的是有 bug），所以可以用这两个选项来指定 PCRE、OpenSSL 的源码目录，从而使用最新的版本，例如：

```
./configure \ #定制 Nginx 的模块
--with-pcre=/path/to/pcre-8.33 #使用 PCRE 8.33 的源码
--with-openssl=/path/to/openssl-1.0.2d #使用 OpenSSL 1.0.2d 的源码
```

--with-xxx_module, --without-xxx_module:

这是一系列近百个模块配置选项，用来决定在 Nginx 里启用或者禁用哪些自带的功能模块，我们可以根据自身的实际情况来定制 Nginx 的功能，例如：

```
./configure \ #定制 Nginx 的模块
--with-http_ssl_module \ #支持 HTTP SSL
--with-http_v2_module \ #支持 HTTP 2.0
--without-http_fastcgi_module #不使用 fastcgi
```

① configure 脚本与 Nginx 开发关系不是太大，通常很少改动，故本书不解析它的实现细节。

② --prefix 参数的作用类似运行时的参数 -p，但它是在编译时确定的，而 -p 可以在运行时改变。

--build=NAME:

这个选项是 Nginx 1.7 版之后增加的新功能，可以在 Nginx 的版本信息里加入自定义的字符串信息，比如公司名称、构建日期、源码版本号等，让 `nginx -v/-V` 更加可读。例如，下面的配置命令为 Nginx 增加了构建用户名和构建日期：

```
./configure \ #定制构建信息
--build="${USER} build at `date +%Y%m%d`"
```

这条命令在字符串里使用 Shell 直接执行了 `date` 命令，并把结果输出到了 `--build` 参数里，编译后用 `nginx -v` 查看版本信息是：

```
nginx version: nginx/1.12.0 (chrono build at 2017xxxx)
```

--with-debug:

这是一个对于我们研究 Nginx 非常有用的编译选项，启用了 Nginx 的调试模式，可以让 Nginx 在运行日志里打印出更多的调试信息。

--with-ld-opt=OPTIONS:

这个选项用来指定编译链接时的额外参数，可以链接其他第三方库，例如使用 `jemalloc` 来替代 GCC 的内存分配函数：

```
./configure --with-ld-opt="-ljemalloc" #链接 jemalloc 库
```

--add-module=PATH:

这是 Nginx 高扩展性的起点，是 `configure` 最重要的选项，指定第三方模块的源码路径。这样 Nginx 就可以把第三方模块像自带模块一样编译进可执行文件（静态链接），从而扩展 Nginx 的能力。我们会在之后的 5.1 节看到它的详细用法。

需要注意的是，`--add-module` 与其他选项不同，在 `configure` 命令行里可以出现任意次，为 Nginx 添加任意数量的第三方模块。

--add-dynamic-module=PATH:

它是自 Nginx 1.9.11 后新增加的选项，与 `--add-module` 功能基本相同，但可以把模块编译为 `*.so` 形式的动态库，为 Nginx 添加了动态模块的能力。

`--add-dynamic-module` 分离了 Nginx 主可执行程序与模块，可以在启动时灵活组合动态加载，方便 Nginx 功能更新。它同时也能够减少编译的时间消耗，使用“`make modules`”只编译变动的模块，而 Nginx 核心代码则无需重新编译。

1.3 配置 Nginx

configure 是对 Nginx 的静态配置，确定 Nginx 的基本运行环境和功能模块，使用简单的命令行参数就可以完成。但 Nginx 的配置绝不只有 configure 这么简单，实际上，配置文件才是 Nginx 的核心，它与 Nginx 的模块化架构是一个互相依存的整体，决定了 Nginx 的进程数量、运行日志、虚拟主机、反向代理、邮件代理、各种请求处理逻辑、优化调整等方面，众多的模块都要依赖配置文件里的指令才能正常工作。Nginx 在启动时将会读取配置文件，根据配置指令调用不同的模块处理，设置它们的运行参数。

由于本书作者并非运维方面的专家，故删繁就简，只介绍作者认为与开发有关、比较常用的指令，有进一步需要的读者可参考官网 Wiki 或者其他相关资料。^①

1.3.1 配置文件格式

Nginx 的配置文件是一个普通的纯文本文件，使用了 Nginx 自定义的一套配置语法，更接近于脚本语言，混合了 Shell、Perl 和 C 的部分特性，要点叙述如下：

- 与 Shell/Perl 相同，使用 # 开始一个注释行；
- 配置指令以分号结束，可以接受多个参数，用空白字符分隔；
- 可以使用单引号或者双引号来定义字符串，允许用 “\” 转义字符；
- 配置指令和参数也可以用引号来指定，特别是当它含有空格的时候；
- 配置块 (block) 是特殊的配置指令，它有一个 {...} 参数且无须分号结束，{...} 里面可以包含多个配置指令，相当于 C 语言里的复合语句；
- 有的配置指令只能出现在特定的配置块里（即语境 Context）；
- 配置块里可以再包含配置块，嵌套层次没有限制，但需符合配置块的语义；
- 可以使用 include 指令包含其他配置文件，支持 “*” 通配符，类似 C 语言；
- 使用 \$var 可以引用预定义的一些变量，增加配置的灵活性；
- 不能识别或错误的配置指令会导致 Nginx 解析失败，无法启动。

下面列出 Nginx 自带的配置文件片段，部分较重要的配置指令用黑体表示：

```
worker_processes 1;                                #设置 worker 进程的数量为 1

events {                                           #events 配置块
```

^① 除 Nginx 配置文件外，要想更好地发挥 Nginx 的实力可能还需要调整操作系统的内核参数，例如 net.core.somaxconn、sys.fs.file_max 等。

<code>worker_connections 1024;</code>	#worker 的最大连接数
<code>}</code>	#events 配置块结束
 <code>http {</code>	#配置 HTTP 服务, 是 Nginx 的重点
<code>include mime.types;</code>	#包含 mime.types 文件
<code>default_type application/octet-stream;</code>	#指定默认的 MIME 类型
 <code>server {</code>	#server 配置块, 虚拟主机
<code>listen 80;</code>	#监听 80 端口
<code>server_name localhost;</code>	#主机名称
 <code>location / {</code>	#location 配置块
<code>root html;</code>	#设置 http 请求的根目录
<code>index index.html index.htm;</code>	#设置默认的 index 文件
<code>}</code>	#location 配置块结束
 <code>error_page 500 502 503 504 /50x.html;</code>	#设置错误返回页
 <code>location = /50x.html {</code>	#错误返回页 location 配置块
<code>root html;</code>	#设置 http 请求的根目录
<code>}</code>	#location 配置块结束
<code>}</code>	#server 配置块结束
<code>}</code>	#http 配置块结束

这个配置文件片段在全局域里出现了一个配置指令 `worker_processes`, 然后是两个大配置块: `events` 和 `http`, 分别配置了 `event` 模块和 `http` 模块的行为。

`events` 配置块很简单, 里面只有一个 `worker_connections` 指令, 确定每个 `worker` 进程可以处理的最大连接数。

`http` 配置块比较复杂, 它先用 `include` 包含了另一个配置文件 `mime.types`, 设置了默认的 MIME 类型, 然后使用 `server` 块定义了一个端口为 80 的虚拟主机。`server` 块里又有两个 `location` 块, 定义了访问虚拟主机上不同路径时的具体行为, 在这里只指定了文档根目录和设置 `index` 文件。

接下来我们就来了解一些常用的 Nginx 配置指令。

1.3.2 进程配置

以下六个进程配置指令不属于任何配置块, 只能在全局域 (`main`) 里配置。

`worker_processes number | auto;`

设置 Nginx 能够启动的 `worker` 进程的数量, 它直接影响 Nginx 的性能。通常当 `worker`

数与服务器的 CPU 核心数相等时可以获得最佳的性能，这时每一个 worker 都会工作在一个独立的 CPU 核心上，完全消除 CPU 调度的成本（需配合 `worker_cpu_affinity` 指令）。

`worker_processes` 的默认值是 1。如果不清楚服务器的 CPU 核心数量，那么可以设置为 `auto` 参数，Nginx 会尝试探测数量并设置。^①

```
master_process on | off;
```

决定是否启用 Nginx 的进程池机制，默认值是 `on`。如果设置为 `off`，那么 Nginx 不会建立 master 进程，只会用一个 worker 进程处理请求，`worker_processes` 指令也会失效，并发处理能力大大下降。在生产环境中不建议设置为 `off`，但对于研究来说它很有用，可以简化我们的调试工作。

```
daemon on | off;
```

决定是否以守护进程的方式运行 Nginx，默认值是 `on`。大多数情况下 Nginx 应该是一个守护进程，运行在后台，不与终端有任何的交互，也减少了无谓的资源消耗。但禁用守护进程同样有利于我们的研究调试，可以直接用 `cout` 或 `printf` 输出调试信息，而不必去查看日志。

本书中配置 `master_process` 为 `off`，偶尔会配置 `daemon off`。

```
worker_cpu_affinity auto [cpumask];
```

指定 worker 进程运行在某个 CPU 核心上，即 CPU 绑定，对于多核心的 CPU 来说可以减少 CPU 的切换，提高 cache 命中率，让 Nginx 更充分地利用 CPU 资源。

例如，如果在一个 4 核 CPU 上，可以设置如下：

```
worker_processes      4;                #启动 4 个 worker 进程
worker_cpu_affinity 0001 0010 0100 1000; #分别运行在 CPU0/1/2/3 上
```

早期（1.9.10 之前）我们只能使用掩码的方式手工绑定，现在则可以使用 `auto` 参数让 Nginx 自动绑定 CPU。

```
working_directory path;
```

配置 Nginx 的工作目录，实际上仅用来存放 `coredump` 文件，在 Nginx 发生意外崩溃时可以用 `gdb` 调试查找原因。

① 系统内的逻辑 CPU 数量可以用命令 `cat /proc/cpuinfo | grep processor` 查看。

```
worker_shutdown_timeout time;
```

这是 1.11.11 版新增的指令。当使用“-s quit”要求终止运行时，Nginx 将最多等待 *time* 的时间，然后强制关闭进程。它可以较好地解决系统里大量出现“is shutting down”状态 Nginx 进程的问题。

1.3.3 动态模块配置

Nginx 在 1.9.11 后支持动态加载模块，指令格式是：

```
load_module file;
```

file 参数标记了模块的文件名，可以指定绝对路径，也可以用相对路径，默认查找路径是安装后的“/usr/local/nginx”目录（即--prefix 或 -p 指定的路径），例如：

```
load_module modules/nginx_XXX_module.so;      #运行时动态加载模块 ngx_XXX_module
```

注意 `load_module` 指令只能在全局域里配置，而且必须在 `events`、`stream`、`http` 等配置块之前出现，否则 Nginx 会报错，提示“load_module directive is specified too late”。

目前 Nginx 硬性规定最多只能加载 128 个动态模块。

1.3.4 运行日志配置

日志是 Web 服务器非常重要的数字资产，它记录了服务器运行期间的各种信息，可以用来排查故障或者数据分析。

在 Nginx 里运行日志分为两种：记录 TCP/HTTP 访问请求的 `access_log` 和记录服务器错误信息的 `error_log`。本书只使用 `error_log`，它可以在任意域里配置。

```
error_log file|stderr level;
```

指定 Nginx 的运行错误日志，默认是安装目录下的 `logs/error.log`。我们也可以自由设置其他路径，或者使用标准错误输出 `stderr`。第二个参数 `level` 是日志的允许输出级别，取值是 `debug|info|notice|warn|error|crit|alert|emerg`，只有高于这个级别的日志才会记录下来，默认值是 `error`。

如果配置为最低的 `debug` 日志级别，那么在 `configure` 时必须使用“--with-debug”。

1.3.5 events 配置

Nginx 采用事件驱动，利用操作系统内核提供的 `epoll`、`kqueue` 等系统调用来高效地处理网络连接，`events` 配置块就是用来配置 Nginx 的事件机制。

虽然 `events` 配置很重要，但配置指令却并不多，而且默认设置就可以工作得很好。

use method;

配置 Nginx 使用的事件处理方式，在 Linux 下可以选择 `select`、`poll` 和 `epoll`，其中 `epoll` 是最高效的，这也是 Nginx 的默认设置，所以通常无需特意使用 `use` 指令。

accept_mutex on | off;

是否启用进程间的负载均衡机制。早期这个指令的默认值是 `on`，虽然可以让多个 `worker` 进程的工作量更均匀，但因为锁的成本较高，很影响效率，所以 1.11.3 版之后 Nginx 把默认值改成了 `off`，也就是不启用负载均衡机制。^①

与 `use` 指令一样，不建议在配置文件里显式写出这个指令，而且最好把它关闭 (`off`)。

worker_connections number;

设置每个 `worker` 进程可以处理的最大连接数量，它决定了 Nginx 的并发能力。注意此指令只是单个进程的处理能力，Nginx 的整体的最大可处理连接数应该再乘上 `worker_processes` 的数量。例如，如果 Nginx 有 10 个 `worker`，每个 `worker` 的连接数是 1 万，那么 Nginx 理论上最多可以支持 $10 \times 1 \text{ 万} = 10 \text{ 万个}$ 并发连接。

`worker_connections` 的默认值是 1024，读者可根据实际情况适当增大。

1.3.6 http 配置

Nginx 近 80% 的功能都是提供 HTTP 服务，所以 HTTP 的配置也是最复杂的。从 1.3.1 节可以看到，`http` 配置块几乎占据了整个配置文件，而实际运行环境里的配置还会更加庞大。

Nginx 使用 `http` 块配置 HTTP 相关的所有功能，包括 `cache`、`fastcgi`、`gzip`、`server`、`location`、`proxy`、`upstream` 等，通常的形式是：

```
http {                                     #http 配置块开始，所有的 HTTP 相关功能
    upstream {                             #upstream 配置块，配置上游服务器
```

^① 现在 Nginx 还可以使用内核级别的负载均衡技术，如 `EPOLLEXCLUSIVE` 或 `reuseport`。

```

    ...
}                                     #upstream 配置块结束

server {                             #server 配置块, 第一个虚拟主机
    listen 80;                       #监听 80 端口

    location / {                     #location 配置块
        ...
    }                               #location 配置块结束
}                                   #server 配置块结束

server {                             #server 配置块, 第二个虚拟主机
    ...
}                                   #server 配置块结束
}                                   #http 配置块结束

```

由于 http 块内容太多, 如果都在一个文件里配置会造成配置文件过度庞大, 难以维护。在实践中我们通常把 server、location 等配置分离到单独的文件, 再利用 include 指令包含进来, 这样就可以很好地降低配置文件的复杂度。

使用 include 后 http 块就简化成了:

```

http {                               #http 配置块开始, 所有的 HTTP 相关功能

    include common.conf             #基本的 HTTP 配置文件, 配置通用参数

    include upstream.conf           #包含 upstream 配置文件, 配置上游服务器

    include servers/*.conf          #包含 servers 目录下所有虚拟主机配置文件
}                                   #http 配置块结束

```

在 http 块里常用的有下面三个指令, 配置 HTTP 服务的通用功能。

```
resolver address ... [valid=time] [ipv6=on|off];
```

这是个比较重要的指令, 配置域名解析服务器, 否则 Nginx 将无法正确解析域名的 IP 地址, 也就无法访问后端的 Web 服务。

```
keepalive_timeout timeout; ①
```

设置 keepalive 的超时时间, 默认是 75s。它通常有利于客户端复用 HTTP 长连接, 提高服务器的性能。如果希望服务器发送完数据后能够主动断连, 就可以把它设置为 0。

① keepalive_timeout 指令也可以出现在 server 和 location 作用域里。

```
access_log path [format [buffer=size] [flush=time] [if=condition]];
```

`access_log` 指令用于配置 http 的访问日志，日志的格式由 `log_format` 决定。为了优化磁盘读写，可以设置 `buffer` 和 `flush` 选项，指定写磁盘的缓冲区大小和刷新时间。

下面的配置使用了 8KB 的缓存，每 1 秒刷新一次：

```
access_log /var/logs/nginx/access.log buffer=8k flush=1s;
```

1.3.7 server 配置

在 `http` 块内使用 `server` 指令定义一个虚拟主机，它必须是一个配置块，在块内部再使其他用指令来确定虚拟主机的端口、域名等参数，然后 Nginx 就可以对外提供 Web 服务。

```
listen port;
```

`listen` 指令设置虚拟主机监听的端口，默认是 80。实际上 `listen` 指令还有很多参数，可以设置 IP 地址、SSL、`rcvbuf/sndbuf`、HTTP 2.0 支持等，因与本书关系不大故从略。

通常来说，联合使用 `backlog`、`deferred`、`fastopen` 和 `reuseport` 这几个参数就能达到较好的优化效果。^①

```
server_name name ...;
```

`server_name` 指令设置虚拟主机对外提供服务的主机名称，允许使用 “*” 通配符和 “~” 开头的正则表达式。例如 “`www.nginx.org`”、“`*.image.nginx.org`”，默认值是空字符串 “”。当 Nginx 处理请求时将会检查 HTTP 头部的 `Host` 域，选择与 `server_name` 匹配的 `server` 块提供服务，从而达到在一个 Nginx 里实现多个虚拟主机的目的。

对于我们自己的开发研究来说，可以直接使用 `localhost` 或者简单的通配符 `*.*`，用类似 “`curl -v http://localhost/...`” 这样的命令就能够访问 Nginx 服务。

```
server_tokens on | build | off;
```

`server_tokens` 指令控制 HTTP 响应头里的 `Server` 字段，`on` 参数会显示 Nginx 版本号，`build` 参数则可以再追加显示编译信息（即 “`--build`” 的字符串）。`off` 用来关闭显示，隐藏具体的版本信息，增强安全性。

这个指令在商业版本的 Nginx Plus 里还可以在配置文件里直接定制显示的字符串，不

① 有的低版本 Linux 可能不支持所有优化参数，例如 `reuseport` 就需要 3.9 以上内核才能使用。

过我们在熟悉了 Nginx 的内部工作原理后也可以在开源版本里添加同样的功能。

1.3.8 location 配置

location 相当于虚拟主机上的虚拟目录，Nginx 在成功匹配虚拟主机进入 server 块后，会继续查找匹配 URI 的 location 块，它是 Nginx 处理的终点站，决定了请求应该如何处理。

location 是一个配置块，但语法稍多一些，除{...}外还有其他的参数：

location [= | ~ | ~* | ^~ | @] uri { ... }

location 使用配置文件里的 uri 参数匹配 HTTP 请求行里的 URI，默认是前缀匹配，也支持正则表达式。但需要注意，如果 uri 里含有“}”、“;”这样的特殊字符（与 Nginx 配置语法冲突）就必须使用单引号或双引号限定，例如：

```
location /image/      {...}      #匹配/image/001.jpg
location ~ '^/\d{2,3}$' {...}      #正则表达式，匹配 2-3 个数字形式的 URI
```

location 使用几个前缀来做进一步的匹配限定：

- = : URI 必须完全匹配；
- ~ : 大小写敏感匹配；
- ~* : 大小写不敏感匹配；
- ^~ : 匹配前半部分即可；
- @ : 用于内部子请求，外部无法访问。

在 server 块里可以配置任意数量的 location 块，location 也可以嵌套。Nginx 对 location 的顺序没有特殊要求，并不是按照配置文件里的顺序逐个查找匹配，而是对所有可能的匹配进行排序，查找最佳匹配的 location。

不同的 location 里可以有不同的处理方式，灵活设置 location 能够让 Nginx 配置清晰明了，易于维护。比如，我们可以在一个 location 里存放静态 html 文件，在另一个 location 里存放图片文件，还有一个 location 则调用 fastcgi 处理 PHP 请求，这些 location 互不干扰，修改其中的一个不会影响其他的正常运行。例如：

```
location /doc/      {...}      #匹配/doc/*. *
location ~ /\. (php)$ {...}      #大小写敏感处理 php 请求
location ~* /\. (png)$ {...}      #忽略大小写，匹配所有的 png 文件
location ^~ /image/ {...}      #匹配/image/*. *，优先级比上一个低
location = /50x.html {...}      #只处理/50x.html 这一个文件
location /          {...}      #匹配任意的 URI
```

需要注意最后一个“location /”，根据 Nginx 的前缀匹配规则，它能够匹配任意的 URI，所以可以把它作为一个“黑洞”，处理所有其他 location 不能处理的请求。

如果 location 配置很多，我们同样可以用 include 的方式来简化配置。

1.3.9 file 配置

在经过了虚拟主机 server 和虚拟目录 location 后，我们需要确定 URI 的处理方式。如果把 Nginx 用作静态 Web 服务器，那么文件访问配置就很简单，只需指定存放路径和文件名即可，这里仅介绍三个最基本的指令，它们可以出现在 http 块里的任何位置。

root path;

设置请求文档的根目录，将以 path 作为起始路径查找文件。如果有：

```
location /image/ {                                #匹配/image/*.  
    root /var/data/;                             #文档根目录实际上是/var/data/  
}
```

那么请求/image/001.jpg 将会返回文件/var/data/image/001.jpg。

对于 1.3.1 节来说，两个 location 文档根目录都是安装目录下的 html 目录。

alias path;

alias 指令同样设置文档的访问目录，但与 root 略有不同，它会把 location 的路径替换为 path，即 location 是 path 的别名。

把刚才的 location 配置改为 alias：

```
location /image/ {                                #匹配/image/*.  
    alias /var/data/;                             #image 路径会映射到/var/data/  
}
```

请求/image/001.jpg 将会返回文件/var/data/001.jpg。

index file ...;

设置 index 文件，即没有指定明确的文件名时的默认文件。

1.3.10 upstream 配置

upstream 块不属于虚拟主机，只能在 http 块里配置，它定义了反向代理时 Nginx 需要访问的后端服务器集群和负载均衡策略。

upstream 块的基本形式是:

```
upstream back_end {                                #upstream 需要有一个名字
    ip_hash;                                       #负载均衡策略
    server 127.0.0.1:80;                          #一台上游服务器
    server ... weight=3;                          #可以指定多台上游服务器
    server ... backup;                            #备份用的上游服务器
}
```

upstream 块的配置比较简单, server 指令指定上游的服务器域名或 IP 地址, 还可以用 weight/max_fails/down/backup 等附加参数来进一步描述服务器的状态。ip_hash 指令确定了这些服务器的负载均衡策略, 如果不给出明确的策略, Nginx 就使用简单的加权轮询 (round robin)。

upstream 块通常配合 proxy_pass、fastcgi_pass 等反向代理指令使用, 把客户端的请求转发到后端的服务器集群处理, 例如:

```
location /passto {                                #一个转发的 location
    proxy_set_header Host $host;                 #转发原始请求的 host 头部
    proxy_pass http://back_end;                  #转发到 upstream 块定义的服务器集群
}
```

本书并不涉及 Nginx 的反向代理服务搭建, 故不对它做详细介绍, 但之后的章节会开发与 upstream 相关的反向代理和负载均衡模块。

1.3.11 变量

如果 Nginx 的配置文件仅有以上的功能, 那么它还称不上是强大。Nginx 的真正神奇之处是可以在配置文件里 (以及 Nginx 运行的任何时刻) 使用内置的 TCP/HTTP 请求相关变量, 并且能够在运行时根据变量值动态变化配置, 使编写配置文件更像是编写 Shell 或者 Perl 程序。

在配置文件里使用 Nginx 变量需要以“\$”开头, 例如\$request_method、\$args、\$uri、\$content_length, 这与 Shell 和 Perl 是一样的。变量的值都是字符串, 可以用在 access_log 里记录访问日志, 用在 proxy_pass 里设置参数, 或者做一些简单的条件判断, 不过变量最大的作用还是供 Nginx 模块获取各种运行时信息。

以下是一些比较常用的 Nginx 变量:

- \$nginx_version: 当前 Nginx 的版本号;
- \$uri : 当前请求的 URI, 但不含“?”后的参数;
- \$is_args : 当前请求是否带参数, 如有则参数值为“?”, 否则是空字符串;

- `$args` :当前请求的参数, 即“?”后的字符串;
- `$request_uri` :当前请求的完整 URI, 包含参数, 相当于`urisis_args$args`;
- `$arg_xxx` :当前请求里的某个参数, “arg_”后是参数的名字;
- `$http_xxx` :当前请求里的 `xxx` 头部对应的值;
- `$sent_http_xxx`:返回给客户端的响应头部对应的值;
- `$remote_addr` :客户端 IP 地址;
- `$limit_rate` :客户端连接速率限制。

如果执行下面的 `curl` 命令:

```
curl -vo /dev/null 'http://localhost/index.html?a=1&b=2' -H 'hello: world'
```

那么在 Nginx 里这些变量的值就是:

```
$nginx_version      = 1.12.0
$uri                 = /index.html
$sis_arg             = ?
$args                = a=1&b=2
$request_uri         = /index.html?a=1&b=2
$arg_a               = 1
$arg_b               = 2
$http_hello          = world
$sent_http_server    = nginx/1.12.0
$remote_addr         = 127.0.0.1
```

在内置的变量以外, Nginx 也允许我们自定义变量, 下面介绍两个指令。

set \$variable value;

`set` 指令相当于`$variable=value`, 即令`$variable`的值为`value`, 是一条基本的赋值语句, 但需要注意变量名不要和已有的内置变量名冲突。

`set` 的用法比较简单, 结合内置变量可以组合出很多新的变量, 例如:

```
set $max_size      10000;           #定义变量$max_size="10000"
set $new_uri        /v2$request_uri #给$request_uri 增加了一个前缀
set $log_tag        "extra action"  #给日志用的一个特别标记
```

map string \$variable { ... }

`map` 指令只能出现在 `http` 块内, 可以使用简单的条件逻辑把 `string` 值转换后赋值给 `$variable`, 在`{...}`里用 `default` 设置默认值, 用通配符或者正则表达式匹配做分支处理, 有些类似 C 语言的 `switch` 语句。

我们使用一个简单的例子来理解 map 的用法：

```
map $sis_args $my_flag {  
    default      0;           #取$sis_args 的值，定义$my_flag  
    "?"         1;           #默认值是 0  
}
```

这样，我们使用 map 指令就把 \$sis_args 变量由字符串转换成了数字。

变量属于 Nginx 里比较高级的部分，用法灵活丰富，可以让 Nginx 在不使用外部编程语言的情况下实现很多高级功能，限于本书的范围，更多的用法不再讨论。

1.4 总结

本章首先介绍了 Nginx 的历史和特点，然后讲解了如何在 Linux 上以源码的方式安装 Nginx，最后简要地阐述了 Nginx 的配置文件格式和里面的各个组成要素。

如果把 Nginx 比作是一座辉煌的宫殿，那么本章就是从远处眺望这座宫殿，看到了它的大致轮廓和外貌，还没有走进去细致地观察。

通过学习本章，读者应该对 Nginx 有一个基本的了解，能够凭自己的力量搭建 Nginx 服务器，实际动手操作无疑能够加强学习的效果。

再提醒读者一下，本章仅仅是“Nginx 入门”，并没有也不可能完整介绍 Nginx 的编译、安装和配置选项，很多重要的功能例如缓存、访问控制、重定向、反向代理等都没有涉及，要想更好地管理维护 Nginx 必须要参考 Nginx 官网或者其他资料。

第 2 章

Nginx开发准备

本章是正式编码开发 Nginx 前的准备工作，将简要介绍 Nginx 的源码目录结构、代码风格 and 特点，以及使用 C/C++ 开发模块的初步技术，有了这些基本知识才能进行后续的 Nginx 模块开发。

2.1 开发环境

第 0 章的 0.4 节介绍了本书的运行环境，但使用 C++ 开发 Nginx 模块还需要明确两个要素：C++ 标准和 Boost 程序库。

2.1.1 C++ 标准

C++ 是 C 的不完全超集，自 1979 年诞生以来一直是系统级开发的主要语言。目前 C++ 有四个国际标准，分别是 C++98、C++11、C++14 和 C++17，其中 C++98 过于陈旧，而 C++14/17 又太新，故本书采用目前支持比较完善的 C++11。

C++11 包含很多友好的新特性，本书使用的有：

- `final` : 显式禁止类被继承；
- `default` : 显式实现默认构造函数和析构函数；
- `nullptr` : 强类型的空指针，用来代替 C 语言里的宏 `NULL`；
- `auto/decltype` : 自动类型推导，可以写出类似 Lua/Python 的简洁代码；
- `lambda` 表达式 : 就地编写匿名函数（闭包），非常灵活易用；
- 可变参数模板 : 可以在模板参数列表里书写任意数量的类型名。

为了使用 C++11 标准，读者需要采用较新版本的 GCC，至少要高于 4.6。可能的操作

系统自带的 GCC 未能达到这个要求, 请参考网络上相关文章升级 GCC, 作者也编写了一个简单的 CentOS 升级 Shell 脚本并发布在 GitHub 上, 见 0.7 节。^①

有了 C++11 标准还不够, 我们还需要另外一个强大的工具——Boost 程序库。

2.1.2 Boost 程序库

Boost 程序库是一个功能强大、构造精巧、跨平台、开源并且完全免费的 C++ 程序库。它由 C++ 标准委员会部分成员所设立的 Boost 社区开发并维护, 使用了许多现代 C++ 编程技术, 涵盖字符串处理、容器与数据结构、泛型编程等许多领域, 是对 C++ 标准极好的补充和完善, 有着“C++ 准标准库”的美誉。

本书使用的 Boost 程序库是 1.62 版, 当然读者也可以采用更新的版本。

Linux 等操作系统通常不会自带 Boost 程序库 (或者版本较旧), 需要我们在官网 <http://www.boost.org> 下载源码包自行编译安装。例如下载 `boost_1_62_0.tar.gz`, 解压缩后进入目录再执行命令:

```
./bootstrap.sh                #配置编译选项
sudo ./b2 link=static install  #编译静态链接库并安装
```

这样即可把 Boost 程序库安装到“/usr/local/”目录下。^②

2.2 目录结构

Nginx 的目录结构简单清晰, 功能和层次都很明确, 非常利于研究和学习。

Nginx 的主目录结构如下:

~/nginx	#源码包解压缩后的根目录
— auto	#configure 编译相关的脚本
— conf	#默认的配置文件的, 示例用
— contrib	#一些实用工具, 如 vim 的语法高亮配置
— html	#默认的 html 文件, 示例用
— man	#用于 UNIX 的 man 帮助文件
— objs	#执行 configure 后产生的目录
— src	#Nginx 的所有实现源码

① 目前 GCC 的最新版本是 6.x, 但本书仍然使用 4.8.x。

② 这里介绍的是安装 Boost 程序库最简单的方式, 更详细知识可参见附录 A 推荐书目 [3][4]。

对于我们模块开发者来说，最需要关心的是 `auto` 目录和 `src` 目录。

`auto` 目录存放了很多 Shell 脚本，被 `configure` 调用，它的结构是：

```
~/nginx/
├── auto
│   ├── cc
│   ├── lib
│   ├── os
│   └── types
```

#源码包解压缩后的根目录
#configure 编译相关的脚本
#检查各种编译器的脚本
#检查各种编译依赖库的脚本
#检查各种操作系统的脚本
#检查平台相关的基本类型的脚本

`src` 目录是我们需要重点关注的目录，所有的 Nginx 实现源码都存放在这里，并且依据模块类型被进一步分门别类：^①

```
~/nginx/
├── src
│   ├── core
│   ├── event
│   │   └── modules
│   ├── http
│   │   ├── modules
│   │   └── v2
│   ├── mail
│   ├── misc
│   ├── os
│   │   └── unix
│   └── stream
```

#源码包解压缩后的根目录
#Nginx 的所有实现源码
#基本的数据结构定义和核心代码
#event 模块的代码
#各种事件驱动模型的具体实现代码
#http 模块的代码
#大量的官方 http 功能模块代码
#处理 HTTP2.0 的功能模块
#mail 模块的代码
#一些辅助代码
#操作系统相关的代码
#UNIX/Linux 系统相关的代码
#流处理功能，即 TCP/UDP 协议处理

在 `src` 里比较重要的目录是 `core`、`http`、`stream` 和 `os/unix`，它们分别存放了 Nginx 的核心代码、HTTP 功能代码、TCP/UDP 功能代码和操作系统相关的代码。

此外，`objs` 目录里还有一些由 `configure` 脚本自动生成的源码文件，定义了 Nginx 静态模块数组和各种常量宏，虽然简单但也是 Nginx 的重要组成部分。

在熟悉了 Nginx 的源码目录结构后，我们就可以有针对性地去阅读源码研究它的工作原理，比如想修改编译配置选项就看 `auto` 目录，想学习 Nginx 整体架构就看 `src/core` 目录，想了解具体的 `http` 模块功能就看 `src/http` 目录，想直接处理 TCP/UDP 协议就看 `src/stream` 目录。

^① 从 Nginx 的 1.9.0 版才开始有 `src/stream` 目录。

2.3 源码特点

本节简要分析 Nginx 的代码风格和内在的编程思想,有助于读者从总体上把握 Nginx 源码,在阅读程序时留意这些特点可以更快地理解代码。

2.3.1 代码风格

Nginx 完全以纯 C 语言实现,基本上遵循了传统的 K&R 风格,与 Linux 等很相似。

在显而易见的单词对齐、空白字符的使用之外,Nginx 源码的特点还有:

- 变量名、函数名全小写,宏全大写(但也有例外),使用下划线连接;
- 类型名、函数名使用前缀“ngx_”,宏使用前缀“NGX_”;
- 结构体(struct)使用后缀“_s”,同名的“_t”后缀是它的等价形式;^①
- 使用 C 的“/**/”注释,不使用新的“//”风格注释。

2.3.2 代码优化

Nginx 使用了很多优化技巧来达到节约系统资源、提高运行效率的目的,例如:

- 自定义一些整数类型实现跨平台编译运行,如 ngx_uint_t;
- 标志位变量使用了不太常用的“位域”特性(bit field),减少内存占用;
- 使用“哨兵”变量(即空对象)表示数组的结束,简化了循环的范围检查,例如 ngx_null_command;
- 使用宏封装了一些简单的操作,消除函数调用的成本。注意:因为这些宏相当于函数,所以采用了小写的形式,本书之后称它们为“函数宏”。

2.3.3 面向对象思想

面向对象是现代软件设计的共识,Nginx 当然也不例外,模块化架构就是最明显的证据。但受限于 C 的语法,Nginx 虽然使用了面向对象的思想但表现方式上却显得比较“委婉”,采用了许多“不得已而为之”的变通之举:

- 没有继承概念,只能使用 void* 指针成员,然后每个“子类”再具体化自己的数据结构,相当于自行实现了 C++ 的对象内存模型;

^① 在 C 语言中,struct 是“二等公民”,类型前必须用 struct 关键字,所以 Nginx 的解决方案是定义一个 xxx_s 的 struct,再 typedef 为 xxx_t 来简化使用方式。

- 没有 RTTI（运行时类型信息），只能用一个 tag 成员来标记类型；
- 没有成员函数，只能用原始的函数指针作为结构的成员；
- 没有模板和泛型，解决方案同样是 void* 指针，因为它可以转型为任意类型的数据，但这也造成了理解和使用上的困难；
- 没有 C++ 里的引用概念，因此要修改值时只能使用函数形式的宏；
- 不支持函数重载，所以有很多实现相同功能、名称近似的函数族；
- 大量使用宏（小写形式）来实现类似成员函数调用的“API”，存在容易误用的隐患。

这些语法上的“局限”使有的 Nginx 的源码片段比较晦涩难懂，增加了使用 C 语言开发 Nginx 模块的难度，但可以使用 C++ 语言来简化。

2.4 使用 C++

C++ 是 C 的“精神继承者”，它高度（但不是百分之百）兼容 C，同时又增加了面向对象、泛型等很多现代编程范式，使用 C++ 可以更容易、更方便地实现良好的软件结构和代码，提高开发效率。

Nginx 受 C 语言限制所产生的所有“代码缺点”都可以在 C++ 中找到对应的解决方案。

2.4.1 实现原则

从设计模式的视角，我们可以把 Nginx 看作一个底层系统，使用包装外观模式 (Wrapper Facade) 把 Nginx 的数据结构和接口函数重新封装整理，在不影响效率的前提下给出一个更易用的面向对象的接口。

本书编写 C++ 代码的一些基本原则是：

- 使用类封装 Nginx 数据结构和相应的操作，增强内聚性；
- 使用 inline 和 static 成员函数消除函数调用的成本，用来替代宏；
- 尽量使用变量的引用形式而不是指针形式；
- 使用函数重载，简化相同功能的函数调用；
- 使用模板和泛型技术消灭 void* 的使用；
- 尽量在编译期把类型信息保存起来，避免运行时的类型转换；
- 使用异常简化错误处理流程；
- 常量宏保留，不做封装（因为改为 enum 并不会带来好处）。

还需要说明的是，出于代码简洁的考虑，本书虽然使用了名字空间特性，但所有的 C++ 类均使用 using 关键字引入到了全局名字空间，无需特意使用名字空间限定。

2.4.2 代码风格

使用 C 语言开发 Nginx 模块应该尽量遵守 Nginx 的编码风格,但使用 C++则不一定要如此,毕竟 C++不是 C,采用不同的编码风格可以更好地区分这两者,也利于代码的维护。

本书使用的 C++代码风格是:

- 变量名和函数名延续 Nginx 风格,全小写,使用下划线连接;
- 类名使用 camel case,即单词的首字母大写;
- 类名均使用前缀“Ngx”,例如 NgxPool;
- 类内部的 typedef 类型名全小写,使用下划线连接;
- 对外的模块变量名和配置指令使用前缀“ndg_”(Nginx Development Guide);
- 在一个 .hpp 源文件里实现该类的所有功能代码;^①
- 使用 C++风格的“//”注释。

2.4.3 编译脚本

Nginx 本身只支持使用 C 语言开发模块,如果我们要使用 C++语言,那么就必须修改源码目录里的编译脚本,也就是 auto 目录里的文件。^②

分析

首先要明确,gcc 不仅能够编译 C 代码,也能够编译 C++代码,但 C 与 C++程序混合编译时应该使用 g++来链接,否则会因为 C 和 C++的编译链接符号不同而链接失败。

所以我们需要使用文件扩展名区分源码(*.c 和 *.cpp),单独给 gcc 增加额外的 C++编译选项来编译我们自己的 C++代码。如果使用其他第三方 C++库(例如 Boost),则可以在 configure 时用--with-ld-opt 参数。

configure 生成的 Makefile 决定了源码的编译链接方式,以此为出发点进行分析梳理就可以找到编译脚本修改的方法。在这里我们略去对 configure 的分析过程,直达结论:产生 Makefile 的 Shell 脚本是 auto/make。

① 本书的 C++代码大都是泛型的,C++技术限制模板类的声明和实现最好在一个文件里。

② 另一种方式是修改运行 configure 后生成的 objs/Makefile,但这样做有很大的缺点——每次 configure 之后都要手动修改,无法做到自动化编译。

修改要点

make 脚本代码并不复杂，有近 700 多行，只要了解一些 Shell 编程知识就可以很容易地为它增添 C++ 编译链接功能，要点是：

- 链接器 LINK 改用 g++；
- 增加 C++ 编译选项，为 gcc 启用 C++11 标准；
- 生成编译命令时用扩展名分支处理 C 源码和 C++ 源码。

修改时还需要注意 Shell 的版本问题，configure 使用系统默认 Shell (/bin/sh)，而不同的系统默认 Shell 可能是不同的，为了保证最大的兼容性，最好不要使用某些 Shell 的特殊语法。

在 make 脚本里添加新的编译选项和链接器如下：

```
#LINK = $LINK                                #原 26 行注释掉原链接器

CXXFLAGS = -std=c++11 -Wall                    #新增 C++ 编译选项，启用 C++11
LINK = g++                                     #新增 C++ 链接器
```

这里可以给 CXXFLAGS 增加更多的参数，例如 -g、-O0 等，读者可根据需要自行增减。

修改静态模块代码

有了 C++ 编译选项和链接器后，我们接着修改 400 行附近的编译静态模块部分，黑体字标明了修改的部分：

```
for ngx_src in $NGX_ADDON_SRCS
do
    ...
    ext=`echo ${ngx_src}|cut -d . -f 2`          #获取源码文件的后缀
    ngx_cxx=$ngx_cc                             #默认不使用 C++ 编译选项
    if [ $ext = "cpp" ]; then                   #如果后缀是 cpp
        ngx_cxx="$ngx_cc ${CXXFLAGS}"          #为 gcc 添加 C++ 编译选项
    fi
    ...

$ngx_obj: \$(ADDON_DEPS)$ngx_cont$ngx_src      #注意要改用 ngx_cxx
$ngx_cxx$ngx_tab$ngx_objout$ngx_obj$ngx_tab$ngx_src$NGX_AUX

done
```

这段脚本里使用了 UNIX 小工具 cut 来获取文件的扩展名，比较简陋。这是因为 Ubuntu 系统的 dash Shell 没有很方便地处理字符串的功能，如果我们使用 bash，那么就可以很简

单地得到扩展名：^①

```
ext=`echo ${ngx_src: (-4)}` #bash 支持直接截取末尾字符串
```

修改动态模块代码

动态模块代码的修改与静态模块的类似，位于 600 行附近，但判断源文件的变量是 `$ngx_source`：

```
ext=`echo ${ngx_source}|cut -d . -f 2` #注意 Shell 变量的名字
ngx_cxx=$ngx_cc #默认不使用 C++编译选项
if [ $ext = "cpp" ]; then #如果后缀是 cpp
    ngx_cxx="$ngx_cc \$(CXXFLAGS)" #为 gcc 添加 C++编译选项
fi

$ngx_obj: $(ADDON_DEPS)$ngx_cont$ngx_src #改用 ngx_cxx
$ngx_cxx$ngx_tab$ngx_objout$ngx_obj$ngx_tab$ngx_src$NGX_AUX
```

其他方法

本节介绍的修改 `auto/make` 脚本是实现 Nginx 支持 C++最简单的方法，但它还不够完善，因为不能定制 C++编译参数。读者可以在熟悉 Nginx 之后改进它，修改 `auto` 目录里的其他脚本，让它工作得更好。^②

2.5 C++包装类

依据上一节的 C++实现原则和编码风格，我们来实现 Nginx 的基本包装类 `NgxWrapper`，它是本书后续很多类的基类。

`NgxWrapper` 是一个模板类，所以不需要知道 Nginx 的任何信息，完全是独立的。

2.5.1 类定义

`NgxWrapper` 应用了代理模式，可以包装代理一个 Nginx 的数据对象，例如 `ngx_str_t`、`ngx_array_t`、`ngx_list_t` 等^③，控制它访问方式，实现代码很像 `boost::reference_wrapper`。

① Shell 编程并非作者强项，也许有更好的实现方法，请读者见谅。

② 提示：可以在 `auto/options` 文件里增加 `--with-cxx-opt` 选项，定义变量 `NGX_CXX_OPT`，然后在 `auto/cc/conf` 或者 `auto/make` 里赋值给 `CXXFLAGS`。

③ 实际上因为 `NgxWrapper` 是泛型的，可以代理任意对象。

NgxWrapper 的定义如下:

```
#include <boost/type_traits.hpp>           //使用 type_traits 元函数计算类型

template<typename T>                       //包装类型 T
class NgxWrapper
{
public:
    typedef typename boost::remove_pointer<T>::type    wrapped_type;

    typedef wrapped_type*                               pointer_type;
    typedef wrapped_type&                               reference_type;
private:
    pointer_type m_ptr = nullptr;                //使用指针保存对象, 默认是空指针
    ...                                           //成员函数见后
};
```

NgxWrapper 是对类型 T 的一层薄薄的封装, 使用一个指针成员 m_ptr 来持有对象, 成本很低。考虑到类型 T 可能也是指针, 而我们实际并不想要保存 T**, 所以使用了 boost.type_traits 库的元函数移除了类型里的指针修饰, 获得原始的待包装类型。^①

2.5.2 构造和析构

NgxWrapper 只能被继承使用, 是一个抽象类, 所以它的构造函数和析构函数应该是被保护的:

```
protected:                                //构造析构只能被子类访问
    NgxWrapper(pointer_type p):m_ptr(p)    //参数是指针类型
    {}

    NgxWrapper(reference_type x):m_ptr(&x) //参数是引用类型
    {}

    ~NgxWrapper() = default;               //析构函数不做任何事
```

为了方便使用, NgxWrapper 的构造函数有两种重载形式, 分别接受指针类型和引用类型, 这样用户可以随意传入变量, 它可以自动选择恰当的方式获取变量指针。

NgxWrapper 不负责对象的生命周期管理, 所以析构函数不删除指针。^②

① 这里我们也可以使用 C++11 标准<type_traits>里的元函数 std::remove_pointer。

② 在 Nginx 里, 对象都是在内存池里统一分配的, 销毁也由内存池负责, 参见第 3 章。

2.5.3 成员函数

由于 NgxWrapper 只有一个指针成员，所以它的访问接口非常简单：

```
public:
    pointer_type get() const           //访问指针成员
    {
        return m_ptr;                 //返回内部保存的指针
    }
```

同样，为了方便使用，NgxWrapper 还重载了转型操作符和指针操作符，可以在需要的时候自动转化为 bool 或者被包装的类型 T，好像是这层包装不存在一样：

```
operator bool () const               //转型为 bool
{
    return get();                     //调用 get()
}

operator pointer_type () const       //转型为指针类型
{
    return get();                     //调用 get()
}

pointer_type operator->() const       //指针操作符重载
{
    return get();                     //调用 get()
}
```

需要注意的是这些成员函数都是 const 的，因为它们不会变动成员变量。

2.6 总结

本章是 Nginx 模块开发前的预备，我们已经站到了这座宏伟的宫殿的大门口，即将登堂入室。

本书主要使用 C++ 语言，所以我们首先了解了开发必备的 C++11 标准和 Boost 程序库，然后学习了 Nginx 的源码目录结构、代码特点和内在的编程思想。针对 Nginx 源码的特点，本章介绍了 C++ 开发的原则和如何修改编译脚本以支持 C++ 编写模块，最后使用 C++ 泛型技术实现了一个基本的包装类 NgxWrapper。

读者可以在接下来的章节里继续体会这些编程风格、实现原则和包装类的应用。

第 3 章

Nginx基础设施

从这里起我们将正式研究 Nginx 源码，学习如何使用 C/C++编写 Nginx 模块。

本章从最基本的数据结构和功能入手，学习 Nginx 里的整数类型、内存池、字符串、时间日期和错误日志等 Nginx 基础设施。这些内容并不直接与 TCP/HTTP 处理相关，但却是 Nginx 开发的必备基础。

本章的代码部分依赖于 Nginx，所以不能简单地在 `main()` 函数里编译运行，心急的读者可以交叉参考第 5 章，了解如何编写一个 Nginx 测试模块。

3.1 头文件

C/C++程序通常都要使用头文件来定义一些基本工具，本书主要使用两个头文件。

3.1.1 Nginx 头文件

因为我们主要是开发 HTTP 功能模块，所以需要关心 `core` 目录和 `http` 目录。Nginx 在 `http` 目录里提供了总括性的头文件 `<ngx_http.h>`，包含了绝大部分必需的宏、类型定义和函数声明。^①

由于 C++与 C 的编译机制不同，我们不能直接包含 `<ngx_http.h>`。要想在 C++程序里使用 Nginx 的 C 代码必须使用 `extern "C"` 指令，这样 C++编译器才能正确链接。

^① 实际上 `<ngx_http.h>` 内部还包含了 `core` 目录里的 `<ngx_config.h>` 和 `<ngx_core.h>`，它们定义了大部分的基本数据结构。

Nginx.hpp 是本书的基准头文件，它包含了开发 Nginx 模块所必需的基本功能代码：

```
// Nginx.hpp
#include <nginx.h>                                //Nginx 版本号定义，在 core 目录

extern "C" {                                     //需使用 extern "C"才能正确链接
#include <ngx_http.h>                             //Nginx 头文件，内含大量定义
}                                                  //extern "C"结束
```

头文件<nginx.h>比较特殊，它只有宏定义，不含 C 结构，所以无需使用 extern "C"。

3.1.2 C++头文件

C++标准和 Boost 程序库提供了很多便利的工具，本书使用 NgxCppInc.hpp 统一引入：

```
// NgxCppInc.hpp
#include <cassert>                                //标准断言
#include <string>                                  //标准字符串
...                                                //其他 C++标准头文件

#include <boost/core/ignore_unused.hpp>           //用于忽略某些不使用的变量
#include <boost/noncopyable.hpp>                  //不可拷贝工具类
...                                                //其他 Boost 程序库头文件
```

本书后续代码里较常用的工具是<cassert>提供标准断言宏 assert，而<boost/core/ignore_unused.hpp>则提供一个有用的模板函数 ignore_unused()，可以忽略暂不使用的变量，用来显式消除编译警告。^①

本节之后的所有代码均包含 Nginx.hpp 和 NgxCppInc.hpp，为节约篇幅不再特别列出。

3.2 整数类型

整数是计算机里最基本的类型，也是处理速度最快的类型。C/C++内置了 short、int、long 等标准整数类型，但这些类型是平台相关的，在不同的系统里精度可能不一样。

为了保证跨平台的兼容性，Nginx 里整数类型的用法很有特点，我们来一起看一下。

^① ignore_unused、noncopyable 等 Boost 工具的用法可参考附录 A 推荐书目[3]。

3.2.1 标准整数类型

Nginx 主要使用了三个标准整数类型：size_t、u_char 和 off_t。

size_t 和 u_char 通常同时出现，表示 Nginx 里的数据块，例如字符串和缓冲区；off_t 则表示文件偏移量。

size_t 用于计算数据的长度，是 C/C++ 标准里定义的一个类型，是 sizeof 操作符的结果类型，相当于 unsigned long。在 64 位的系统里宽度是 8 字节：

```
assert(sizeof(size_t) == 8); //检查 size_t 的宽度
```

u_char 表示一个字节，虽然它也是一个标准整数类型，但并不是 C/C++ 标准，而是“系统标准”，是在 <sys/types.h> 里的一个 typedef：

```
typedef unsigned char u_char; //u_char 的类型定义
```

3.2.2 自定义整数类型

Nginx 依据 C/C++ 标准重新定义了三个整数类型来代替标准里的 int/long 等类型，保证在任何系统上编译都能获得一致的结果。这些类型是：^①

```
// 定义在 core/nginx_config.h
typedef intptr_t      ngx_int_t;           //有符号整数
typedef uintptr_t     ngx_uint_t;          //无符号整数
typedef intptr_t      ngx_flag_t;          //标志整数类型
```

使用这三个基本整数，Nginx 又 typedef 了一些等价类型，用于其他用途：

```
// 定义在 core/nginx_rbtree.h，红黑树的键类型
typedef ngx_uint_t     ngx_rbtree_key_t;
typedef ngx_int_t      ngx_rbtree_key_int_t;
```

```
// 定义在 os/unix/nginx_time.h，毫秒的整数类型
typedef ngx_rbtree_key_t     ngx_msec_t;
typedef ngx_rbtree_key_int_t ngx_msec_int_t;
```

我们在开发 Nginx 模块编写代码时应该尽量使用这些 Nginx 专有的整数类型，不仅是为了与 Nginx 保持风格的一致，更重要的是保证代码的跨平台兼容。

^① intptr_t 和 uintptr_t 是 C/C++ 标准里的两个特殊整数类型，它们是大小足够容纳指针的整数类型，可以简单地理解为指针的整数形式。

3.2.3 无效值

变量的初始化是编程语言里一个重要但又常常会被忽略的问题。在 C/C++ 中，整数类型属于 POD 类型，初始值是“未定义”的，也就是说初始值可能是任意数值，存在安全隐患。

Lua/Python 等语言有 Nil/None 的概念，一个变量如果未初始化，那么它的值就是 Nil 或者 None。Nginx 采用了类似思路的“UNSET”值，使用“-1”表示未初始化。

由于 C/C++ 是强类型语言，单纯的整数-1 不能直接与其他类型比较，需要做类型转换，所以 Nginx 为-1 定义了不同类型转换的宏：

```
// 定义在 core/nginx_conf_file.h
#define NGX_CONF_UNSET                                -1 //通用的无效值
#define NGX_CONF_UNSET_UINT      (ngx_uint_t)        -1 //无符号整数的无效值
#define NGX_CONF_UNSET_PTR       (void *)            -1 //指针类型的无效值
#define NGX_CONF_UNSET_SIZE      (size_t)            -1 //size_t 类型的无效值
#define NGX_CONF_UNSET_MSEC      (ngx_msec_t)        -1 //毫秒类型的无效值
...                                                    //还有更多的“-1”定义
```

有了 UNSET 概念，Nginx 以宏的形式提供了初始化和条件赋值函数^①，基本的形式是：

```
// 定义在 core/nginx_conf_file.h
#define ngx_conf_init_value(conf, default)            \
    if (conf == NGX_CONF_UNSET) {                    \
        conf = default;                               \
    }

#define ngx_conf_merge_value(conf, prev, default)    \
    if (conf == NGX_CONF_UNSET) {                    \
        conf = (prev == NGX_CONF_UNSET) ? default : prev; \
    }
```

同样的，这两种函数还有若干个其他形式，用来操作 ngx_uint_t、size_t、指针等类型，但其内部的逻辑是相同的，代码基本是机械重复，只有少量名称的不同。

3.2.4 C++封装

在 3.2.3 节我们看到，由于 C 语言的限制，要操作一个简单的“-1”，居然要用数十行的代码去实现，而且使用也存在很大的不方便——我们必须明确地知道操作的类型，再选择正确的操作函数，否则就会导致无谓的编译错误。

^① 实际上这些“函数”都是宏，但作用上是函数，故作者有时称它们为函数，请读者不要误解。

C++语言里的泛型技术可以很好地解决这个问题，它把类型识别的重担由程序员转移给了编译器，让编译器自动去选择合适的类型和函数，而且绝对不会出现失误。

无效值的封装

我们使用类 `NgxUnsetValue` 封装 `Nginx` 里的无效值 “-1”，使用模板转型操作函数来自动确定类型，并且用标准转型操作符 `static_cast` 和 `reinterpret_cast` 来增强易读性：

```
class NgxUnsetValue final //final 禁止被继承
{
public:
    template<typename T> //模板函数
    operator T () const //转型到类型 T
    {
        return static_cast<T>(-1); //使用 static_cast 转型-1
    }

    template<typename T> //模板函数
    operator T* () const //转型到类型 T*，即指针类型
    {
        return reinterpret_cast<T*>(-1); //使用 reinterpret_cast
    }
};
```

`NgxUnsetValue` 相当于 `Nginx` 里的 `NGX_CONF_UNSET_XXX` 宏，但它使用了模板技术支持任意类型的转型，比只做字符串替换的宏更加安全。

`NgxUnsetValue` 应该是个单件，可以实现一个静态成员函数，提供一个全局访问点：

```
public:
    static const NgxUnsetValue& get() //获取全局唯一对象
    {
        static NgxUnsetValue const v = {}; //静态变量，空类
        return v;
    }
```

这样，`NgxUnsetValue` 就完全封装了 `Nginx` 的 `UNSET` 概念。

我们还可以再为这个单件起一个“别名”，方便使用：

```
auto&& ngx_nil = NgxUnsetValue::get(); //右值引用，创建别名
```

之后直接使用 `ngx_nil` 就能够获得这个类型安全的“-1”值。

操作函数的封装

类 `NgxValue` 封装了对整数类型的基本操作，除了初始化和条件赋值，还增加了判断是

否无效的操作:

```

class NgxValue final                                //final 禁止被继承
{
public:
    NgxValue() = default;                            //良好的编程习惯
    ~NgxValue() = default;                            //为构造和析构提供默认实现
public:
    template<typename T>
    static bool invalid(const T& v)                  //无效值判断
    {
        return v ==                                //与 NgxUnsetValue 比较
            static_cast<T>(NgxUnsetValue::get());    //静态转型
    }

    template<typename T, typename U>
    static void init(T& x, const U& v)
    {
        if (invalid(x))                            //无效值判断
        {
            x = v;                                  //如果无效则初始化
        }
    }

    template<typename T, typename U, typename V>
    static void merge(T& c, const U& p, const V& d) //条件赋值
    {
        if (invalid(c))                            //如果无效则赋值
        {
            c = invalid(p) ? d : p;                  //检查 p, 无效则赋值为 d
        }
    }
};

```

成员函数 `invalid()` 简化了变量与 `NgxUnsetValue` 的比较, `init()` 和 `merge()` 完全复制了 Nginx 源码的逻辑, 但更清晰易读。^①

注意, 这些成员函数都是静态的, 需要以 `NgxValue::xxx()` 的形式调用。

使用 C++ 的操作符重载特性可以进一步简化无效值判断:

```

template<typename T>
bool operator==(const T& x, const NgxUnsetValue&) //与 ngx_nil 比较

```

① 利用 C++ 的重载函数特性, 我们还可以在 `NgxValue` 里针对其他类型实现特别版本的 `init()`、`merge()` 函数, 例如 3.5 节的 `ngx_str_t`。


```
{
    return NgxValue::invalid(x);           //调用 invalid() 函数
}
```

批量未初始化^①

有时候我们需要把多个值置为明确的未初始化状态，虽然有了 ngx_nil，但仍要编写多条语句逐一赋值，比较麻烦。

C++11 的可变参数模板新特性提供了简化批量赋值语句的可能性。它可以像 printf() 那样接受任意多个参数，采用（编译期）递归的方式解包模板参数列表，自动生成多条语句，与手写的效果完全相同。

unset() 函数的实现代码如下：

```
class NgxValue final
{
    ...                                     //invalid() 等函数
public:
    template<typename T, typename ... Args>   //可变模板参数列表
    static void unset(T& v, Args& ... args)   //注意“...”的用法
    {
        v = NgxUnsetValue::get();           //置为未初始化状态
        unset(args...);                     //递归处理剩余的模板参数
    }

    static void unset() {}                  //递归终结函数
};
```

unset() 函数可以这样使用：

```
ngx_int_t    x = 10;                      //一些 Nginx 变量
ngx_uint_t   y = 20;
void*        p ;                           //指针值未定义

NgxValue::unset(x, y, p);                  //批量未初始化操作
assert(p == ngx_nil);                      //验证指针值无效
```

3.3 错误处理

Nginx 使用传统的错误码处理错误（注意与 HTTP 的响应状态码 200、404 等是不同的），但在 C++ 里使用异常（exception）的方式会更好。因为异常机制可以分离代码逻辑里的正

^① 实际上这也是一种“初始化”，只不过是初始化为一个表示无效含义的值。

常部分与异常部分，使代码的结构更加清晰。^①

3.3.1 错误码定义

Nginx 使用宏定义了七个常用的错误码，类型是 `ngx_int_t`：

```
// 定义在 core/nginx_core.h
#define NGX_OK          0           // 执行成功，无错误
#define NGX_ERROR      -1           // 执行失败，最常见的错误码
#define NGX_AGAIN      -2           // 未准备好，需要重试
#define NGX_BUSY       -3           // 后端服务正忙
#define NGX_DONE       -4           // 执行成功，但还需要有后续操作
#define NGX_DECLINED   -5           // 执行成功，但未做处理
#define NGX_ABORT      -6           // 发生了严重的错误
```

在编写自己的功能函数时通常要使用以上的这七个错误码，如果有必要也可以自定义一些错误码，但必须是负值。

3.3.2 C++异常

C++对异常规定得比较宽松，任何类型都可以作为异常抛出，但最好使用 `class`。这里我们使用 C++标准里的 `std::exception` 和 Boost 程序库里的 `boost::exception` 来实现异常类 `NgxException`。

类定义

`NgxException` 的类定义如下：

```
class NgxException final : public virtual std::exception,
                          public virtual boost::exception
{
public:
    typedef boost::string_ref string_ref_type;
private:
    ngx_int_t      m_code = NGX_ERROR;           // 错误代码
    std::string    m_msg;                         // 错误信息
public:
    NgxException(                                     // 完整的构造函数
        ngx_int_t x, string_ref_type msg):
        m_code(x), m_msg(msg)
```

^① 有的读者可能会顾虑异常的性能，事实上现在的 C++编译器的异常机制已经有了很大的优化，虽然还不能达到零成本，但可以说是足够小，完全不需要担心。

```

{}

NgxException(ngx_int_t x = NGX_ERROR):
    NgxException(x, "")                //委托构造
{}

NgxException(string_ref_type msg):
    NgxException(NGX_ERROR, msg)       //委托构造
{}

virtual ~NgxException() noexcept      //虚析构函数
{}
public:
    ngx_int_t code() const             //获取错误码
    {
        return m_code;
    }
    virtual const char* what() const noexcept override
    {
        return m_msg.c_str();
    }
    ...                                //其他成员函数，见后
};

```

依据 C++ 最佳实践准则，异常应该从 `std::exception` 虚继承，我们还使用了 `boost::exception`，可以让异常容纳更多的信息。^①

`NgxException` 有两个成员变量，存储了错误码和对应的描述信息。构造函数有三种形式，用来简化调用，其中后两个构造函数使用了 C++11 的委托构造新特性。

为了避免字符串拷贝的成本，`NgxException` 的函数参数使用了 `boost::string_ref`，这是一个轻量级的字符串表示，没有拷贝代价，类似 3.5 节的 `ngx_str_t`。

成员函数

异常的通常使用方式是直接用 `throw` 抛出，但这里我们把它封装为一个 `raise()` 函数，表现形式更好：

```

public:
    static void raise(ngx_int_t rc = NGX_ERROR, string_ref_type msg = "")
    {

```

^① `boost::exception` 和 `boost::string_ref` 的详细用法可参考附录 A 推荐书目 [3]。

```
        throw NgxException(rc, msg);           //抛出异常
    }
```

检查错误码和空指针是 Nginx 开发中经常要做的工作。反复出现的 if 语句很麻烦，由于异常处理流程与正常流程是分离的，所以可以轻松写出封装函数 `require()`：

```
public:
    //检查条件是否满足
    static void require(bool cond, ngx_int_t e = NGX_ERROR)
    {
        if(!cond)                               //如果不符合预期则抛出异常
        {
            raise(e);
        }

        //检查错误码，默认要求是 NGX_OK
        static void require(ngx_int_t rc, ngx_int_t x = NGX_OK)
        {
            require(rc == x, rc);                //如果不是 OK 则抛出异常
        }

        //检查空指针，要求指针非空
        template<typename T>
        static void require(T* p, ngx_int_t e = NGX_ERROR)
        {
            require(p != nullptr, e);           //如果是空指针则抛出异常
        }
    }
```

有的时候当判断条件成立时的逻辑更容易书写，所以 `require()` 的“反函数”`fail()` 也很有用：

```
static void fail(bool cond, ngx_int_t e = NGX_ERROR)
{
    if(cond)                                     //如果符合预期则抛出异常
    {
        raise(e);
    }
}
```

3.4 内存池

内存池是大型软件里常用的一种技术，它是对象池模式的具体应用：一次性向系统申请大块的内存，内部自行切割分配使用，最后再一次性归还系统。这种方式减少了系统调用的次数，而且能够很好地避免内存碎片和泄漏。

Nginx 的内存池有它自己的特色，为了提高效率，使用了一些巧妙的设计和技巧，机制

比较复杂。本节仅介绍它的使用方法，其详细实现原理的讲解已经超出了本书的范围。

3.4.1 结构定义

Nginx 的内存池数据结构是 `ngx_pool_t`，代码摘要如下：

```
// 定义在 ngx_core.h
typedef struct ngx_pool_s  ngx_pool_t;           //简化定义

// 定义在 ngx_palloc.h
struct ngx_pool_s {                               //结构体定义
    ...                                           //暂无须关心的内部成员
    ngx_pool_cleanup_t    *cleanup;             //“析构”时的清理动作，见后
    ngx_log_t              *log;                //关联的日志对象
};
```

`ngx_pool_t` 代表了 Nginx 里的内存池概念，Nginx 会为每一个 TCP/HTTP 请求创建一个独立的内存池——也就是 `ngx_pool_t` 对象，在整个请求的处理过程中我们可以在里面任意申请内存使用，不必担心内存的释放问题。当请求结束时 Nginx 会自动销毁 `ngx_pool_t` 对象，释放内存池和它拥有的所有内存。

在 Nginx 开发时我们需要改变以往的思路，尽量避免用 `malloc` 或者 `new` 来动态分配内存，而是在 Nginx 的框架里使用 `ngx_pool_t`，从而获得高性能。

3.4.2 操作函数

`ngx_pool_t` 相关的操作函数有很多，由于 Nginx 框架负责内存池的创建与销毁，故本节只讲解内存的分配与释放，介绍几个最常用的函数：

```
void* ngx_palloc(ngx_pool_t *pool, size_t size);
void* ngx_pnalloc(ngx_pool_t *pool, size_t size);
void* ngx_pccalloc(ngx_pool_t *pool, size_t size);
```

这三个函数相当于标准的 `malloc()`，都是从内存池 `pool` 里获取 `size` 字节大小的内存，返回一个 `void*` 指针，区别是：

- `ngx_palloc()` 使用了内存对齐，速度快，但可能会有少量的内存浪费；
- `ngx_pnalloc()` 没有使用内存对齐；
- `ngx_pccalloc()` 内部调用了 `ngx_palloc`，并且把内存块清零。

大多数情况下我们应该使用 `ngx_pccalloc()`，它更加快速安全。

与 `ngx_palloc()` 对应，可以使用 `ngx_pfree()` 函数释放申请的内存，相当于 C 标准函数

free():

```
ngx_int_t ngx_pfree(ngx_pool_t *pool, void *p);
```

3.4.3 C++封装

ngx_pool_t 存在明显的面向对象特征——有数据结构和应用在它上的一些操作，所以可以封装为 C++ 类。

类定义

我们使用 2.5 节的 NgxWrapper 来实现封装类 NgxPool，定义如下：

```
#include "NgxWrapper.hpp"           //基本包装类
#include "NgxException.hpp"         //异常类

class NgxPool final : public NgxWrapper<ngx_pool_t>
{
public:
    typedef NgxWrapper<ngx_pool_t>  super_type;    //简化类型定义
    typedef NgxPool                 this_type;
    ...                                           //成员函数见后
};
```

NgxPool 的基类是 NgxWrapper<ngx_pool_t>，包装了 ngx_pool_t 结构，因为复用了 NgxWrapper 的代码，所以直接获得了操作符重载和 bool 转型的能力。

构造与析构

NgxPool 使用 ngx_pool_t* 指针构造，并委托给 NgxWrapper：

```
public:
    NgxPool(ngx_pool_t* p) : super_type(p)        //构造
    {}

    ~NgxPool() = default;                          //析构
```

由于 ngx_pool_t 在很多 Nginx 的数据结构里以成员变量 pool 的形式出现，所以可以增加一个模板构造函数自动使用这个成员，方便用户调用：^①

```
template<typename T>
NgxPool(T* x) : NgxPool(x->pool)                //使用类型 T 的 pool 成员
```

^① GitHub 上的实际源代码比较复杂，使用了模板元编程，这里做了适当的简化。

```
{}
```

这里的模板参数 `T` 可以是 `ngx_http_request_t`、`ngx_conf_t` 等含有 `pool` 成员的结构。

基本的内存分配函数

我们主要使用 `ngx_palloc()` 分配内存，首先实现一个基本的内存分配函数 `palloc()`，它可以依据模板参数决定是否抛出异常：

```
public:
    template<typename T, bool no_exception = false>
    T* palloc() const
    {
        auto p = ngx_palloc(get(), sizeof(T));           //分配内存, 注意 auto 的使用

        if(!p)                                           //检查空指针
        {
            if(no_exception)                             //是否允许抛出异常
            { return nullptr; }                          //返回空指针

            NgxException::raise();                       //抛出异常
        }

        assert(p);                                       //断言增强安全性
        return new (p) T();                             //构造 T 对象
    }
```

`palloc()` 使用 `sizeof` 计算类型 `T` 的长度再分配内存，如果分配成功那么就在这块内存上调用 `new`，执行 `T` 的默认构造函数，返回的是一个 `T*` 指针，而不是无类型的 `void*`。

`bool` 模板参数 `no_exception` 用来控制 `palloc()` 是否抛出异常，这是因为有的内存分配操作可能允许返回空指针，执行别的处理逻辑。

成员函数 `alloc()` 和 `alloc_noexcept()` 包装了 `palloc()`，简化了调用方式：

```
public:
    template<typename T>
    T* alloc() const                                     //抛异常版本
    {
        return palloc<T, false>();                     //参数是 false
    }

    template<typename T>
    T* alloc_noexcept() const                           //不抛异常版本
```



```

{
    return palloc<T, true>();           //参数是 true
}

```

扩展的内存分配函数

NgxPool 还提供两个函数，用来分配指定字节数的内存和多参数构造对象：

```

public:
    template<typename T>
    T* nalloc(std::size_t n) const           //分配 n 个字节内存
    {
        auto p = ngx_pnalloc(get(), n);    //使用 ngx_pnalloc

        NgxException::require(p);           //检查空指针

        return reinterpret_cast<T*>(p);      //转型为 T*
    }

    template<typename T, typename ... Args>
    T* construct(const Args& ... args) const //任意数量参数构造对象
    {
        auto p = ngx_pcalloc(get(), sizeof(T)); //使用 ngx_pcalloc

        NgxException::require(p);           //检查空指针

        return new (p) T(args ...);        //转发参数构造
    }

```

更多的成员函数

NgxPool 封装了 Nginx 的内存池对象，所以还可以集成更多的内存池相关操作，例如拷贝字符串、创建 `ngx_array_t`/`ngx_list_t` 等，本书之后会继续完善它。

3.4.4 清理机制

Nginx 框架自动管理内存池的生命周期，当一个请求结束时内存池里的内存会完全归还给系统。但这里还存在问题：内存只是系统资源的一个方面，其他的系统资源（如文件句柄）并不会随着内存池的销毁而一并释放，如果不做特殊的操作就有可能造成资源泄漏。

这种清理机制就是 C++ 里的析构函数思想，在对象销毁时自动调用析构函数，执行对应的资源销毁动作。但 Nginx 使用的是 C 语言，而 C 语言并没有析构的概念，所以 Nginx 实现了自己的清理机制。

实现原理

Nginx 定义了一个保存清理信息的结构 `ngx_pool_cleanup_t`，用来在内存销毁时执行清理动作：

```
// 定义在 ngx_palloc.h
typedef void (*ngx_pool_cleanup_pt)(void *data); //清理函数指针原型
typedef struct ngx_pool_cleanup_s ngx_pool_cleanup_t;

struct ngx_pool_cleanup_s { //清理信息结构体
    ngx_pool_cleanup_pt handler; //清理动作，函数指针
    void *data; //清理所需数据
    ngx_pool_cleanup_t *next; //后续链表指针
};
```

`ngx_pool_cleanup_t` 的成员 `handler` 保存了清理函数，它是一个函数指针，成员 `data` 是一个 `void*` 指针，通常指向需释放的资源。Nginx 会在销毁内存池时把 `data` 传递给 `handler`，即调用 `handler(data)` 完成清理动作。注意，因为 `data` 的类型不做要求，所以我们可以利用 `handler` 执行任意操作，不一定必须是“释放资源”。^①

`ngx_pool_cleanup_t` 还有一个 `next` 指针成员，多个对象可以使用这个指针连接成一个单向链表。当内存池销毁时 Nginx 会逐个执行链表里的清理函数，释放所有的资源。

`ngx_pool_t` 结构里的成员 `cleanup` 是这个链表的头节点：

```
struct ngx_pool_s { //ngx_pool_t 结构体定义
    ... //暂无须关心的内部成员
    ngx_pool_cleanup_t *cleanup; //所有的清理动作在此
};
```

管理清理链表

我们不需要直接操作 `ngx_pool_t::cleanup`，Nginx 提供了专门的函数 `ngx_pool_cleanup_add()`，它会自动维护清理链表，声明是：

```
// 定义在 ngx_palloc.h
ngx_pool_cleanup_t *ngx_pool_cleanup_add(ngx_pool_t *p, size_t size);
```

这个函数使用 `size` 为 `ngx_pool_cleanup_t::data` 分配内存，返回清理信息 `ngx_pool_cleanup_t` 对象，设置它的 `handler` 和 `data`，就可以达到向内存池“注册”清理函数的目的。

^① 这种用法很像 `boost::shared_ptr<void>`。

C++封装

NgxPool 对 ngx_pool_cleanup_add() 的封装如下:

```
public:
    template<typename F, typename T>                //模板参数支持任意类型
    ngx_pool_cleanup_t* cleanup(F func, T* data, std::size_t size = 0) const
    {
        auto p =
            ngx_pool_cleanup_add(get(), size);      //调用 Nginx 函数
        NgxException::require(p);                  //检查空指针

        p->handler = func;                          //设置清理函数

        if(data)                                    //允许直接传入待释放的资源
        { p->data = data; }                          //设置 data 成员

        return p;                                    //返回清理信息对象
    }
```

cleanup() 函数描述了 Nginx 里释放资源的基本流程: 首先获取一个 ngx_pool_cleanup_t 对象, 然后设置它的 handler 和 data。如果不提供 data 对象 (nullptr), 那么用户就需要手工设置。

使用析构函数

对于 C++来说析构函数是才释放资源的正确方法, 但在 Nginx 里释放内存池是 C 调用, 并不会触发析构行为。如果我们自己的类用到了其他资源, 就必须把类的析构函数适配到 Nginx 的清理机制里:

```
public:
    template<typename T>
    static void destory(void* p)                    //适配析构函数符合 Nginx 要求
    {
        (reinterpret_cast<T*>(p))->~T();            //转换 void*, 再调用析构函数
    }

    template<typename T>
    void cleanup(T* data) const                      //重载 cleanup() 函数
    {
        cleanup(&this_type::destory<T>, data);    //注意函数指针的用法
    }
```

静态成员函数 destory() 把 void* 类型转换为 T, 然后显式调用它的析构函数, 这样就符合了 Nginx 对 handler 的要求。由于类自身已经包含了资源的所有信息, 所以我们无须

额外创建 data，直接向 Nginx 的清理链表里添加 handler 和 data 即可。

注意 destroy() 是一个模板函数，所以在写函数指针时必须带上模板参数列表。

3.4.5 C++内存分配器

C++标准库和 Boost 程序库提供了大量的泛型容器，例如 vector、list、unordered_map 等，在开发 Nginx 模块时我们完全可以自由使用。但有一点需要留意，它们并不知道 Nginx 内存池的存在，直接向操作系统申请内存，所以可能会有一些效率的损失。

不过 C++的内存分配机制非常灵活，允许用户自定义内存分配器。通过这个机制，我们就可以让这些泛型容器与 Nginx 内存池完美结合，在高效使用内存的同时获得 C++的便捷。

类定义

NgxAllocator 与 NgxPool 基本实现比较相似，都使用 NgxWrapper 来包装 ngx_pool_t 对象：

```
template<typename T>
class NgxAllocator : public NgxWrapper<ngx_pool_t>
{
public:
    typedef NgxWrapper<ngx_pool_t>    super_type;    //简化类型定义
    typedef NgxAllocator               this_type;

public:
    typedef std::size_t                size_type;     //内部类型定义
    typedef T*                         pointer;
    typedef T                          value_type;

public:
    NgxAllocator(ngx_pool_t* p) : super_type(p)        //构造
    {}
    ~NgxAllocator() = default;                        //析构

    ...                                              //成员函数见后
};
```

内存分配函数

NgxAllocator 同样也需要实现内存分配函数，但它的接口需要遵守 C++对内存分配器的约定，这是与 NgxPool 不同的地方。

在 C++11 标准中自定义内存分配器的工作较以前已经有了很大的简化，我们只需实现 allocate() 和 deallocate() 两个基本的函数：

```

public:
    pointer allocate(size_type n)                                //分配 n 个元素所需的内存
    {
        return reinterpret_cast<pointer>(<
            ngx_pnalloc(get(), n * sizeof(T)); //使用 Nginx 内存池
        )

    void deallocate(pointer ptr, size_type n)                  //“释放”内存
    {                                                            //由 Nginx 负责内存回收
        boost::ignore_unused(n);                                //忽略入口参数 n
        ngx_pfree(get(), ptr);                                  //归还内存池
    }

```

用法

以最常用的 `std::vector` 为例，我们来演示一下自定义内存分配器的用法。

`std::vector` 有两个模板参数，使用 `std::allocator` 作为默认的内存分配器，类声明是：

```

template< class T,                                            //容纳的元素类型
        class Allocator = std::allocator<T>                 //内存分配器
        > class vector;

```

所以，只要改写它的第二个模板参数，替换为我们自己的 `NgxAllocator`，然后在构造函数里传入分配器对象，`std::vector` 就能够使用 Nginx 的内存池，例如：

```

//假设已经有内存池对象 pool
std::vector<char, NgxAllocator<char>> > v(pool); //注意模板参数

```

使用 C++11 的模板别名特性可以简化容器的类型声明：

```

template<typename T>
using NgxStdVector =                                       //C++11 新特性
    std::vector<T, NgxAllocator<T>>;                      //模板别名

NgxStdVector<char> v(pool);                                //使用模板别名创建容器

```

3.5 字符串

HTTP 协议是基于纯文本的协议，所以 Nginx 必须能够正确、高效地处理字符串。Nginx 为此设计了 `ngx_str_t` 结构，它表示 Nginx 里的字符串对象。

3.5.1 结构定义

`ngx_str_t` 并不是一个传统意义上的“字符串”，准确地说，它应该是一个“内存块引用”，定义如下：

```
// 定义在 ngx_string.h
typedef struct {
    size_t      len;           // 一个匿名结构体
    u_char*     data;          // 字符串长度
                                // 字符串所在的地址
} ngx_str_t;                  // 类型定义
```

从代码里可以看到，`ngx_str_t` 的结构非常简单，只是用成员变量 `len` 和 `data` 标记了一块内存区域，并不实际持有字符串内容，非常的轻量级。`data` 成员的类型是 `u_char` 而不是 `char`，意味着它不一定必须以 `'\0'` 结尾，任何数据都可以当作字符串来使用。

这样的设计与 `boost::string_ref` 或者 `std::string_view` (C++17) 非常相似。好处是字符串的操作非常廉价——只有两个整数的成本，不需要复制大量的数据，所以对它的拷贝、修改都非常高效，也节约了内存的使用。

当然这种设计也有缺点。因为 `ngx_str_t` 只是引用内存，所以我们应尽量以“只读”的方式去使用它。在多个 `ngx_str_t` 共享一块内存时要小心，如果擅自修改字符串内容很有可能影响其他的 `ngx_str_t` 引用，导致错误发生。另外一种危险是引用的内存地址可能会失效，访问到错误的内存区域。

在实际的 TCP/HTTP 请求处理过程中，我们通常都是以只读的方式去检查收到的数据，很少修改数据。如果确实要修改字符串或者获取完全拥有权，那么可以向 Nginx 内存池单独申请一块内存，把字符串内容复制过去（但同时也增加了内存拷贝的代价）。

3.5.2 操作函数

Nginx 定义了很多字符串操作函数，本书不可能全部讲解（也无必要），只列出一些常用的函数，读者可阅读 Nginx 源码 (`core/ngx_string.*`) 了解更多的用法。

为了提高运行效率，这些“函数”很多都是宏。

初始化和赋值

Nginx 提供两个初始化函数宏：`ngx_string()` 使用字符串初始化，`ngx_null_string()` 初始化为一个空字符串，因为它们使用了 `{...}` 的形式，所以只能用于赋值初始化：

```
#define ngx_string(str) { sizeof(str) - 1, (u_char *) str }
```

```
#define ngx_null_string    { 0, NULL }
```

运行时设置字符串内容可以使用下面两个函数宏，功能是相同的，注意参数 `str` 必须是指针：

```
#define ngx_str_set(str, text) \
    (str)->len = sizeof(text) - 1; (str)->data = (u_char *) text
#define ngx_str_null(str)    (str)->len = 0; (str)->data = NULL
```

`ngx_string()` 和 `ngx_str_set()` 内部使用了 `sizeof` 计算字符串长度，所以参数必须是一个编译期的字符串“字面值”，而不能是一个字符串指针，否则 `sizeof` 会计算得到指针的长度（8 字节），而不是字符串的实际长度。

这四个函数宏的示例用法如下：

```
ngx_str_t s1 = ngx_null_string;           //初始化为空字符串
ngx_str_t s2 = ngx_string("matrix");      //初始化为"matrix"

ngx_str_set(&s1, "reloaded");              //运行时赋值，注意取地址操作符
ngx_str_null(&s2);                        //运行时置为空字符串
```

下面的代码则是错误的：

```
s1 = ngx_null_string;                     //错误！不是赋值初始化
ngx_str_set(s1, "wrong");                  //错误！没有使用取地址操作符
```

基本操作函数

`ngx_str_t` 只是个普通的字符串，所以标准的 C 字符串函数都能够使用（需要转型为 `const char*`），处理它的 `data` 成员就可以了，但 Nginx 也实现了一些特有的操作函数。^①

下面是一些常用的字符串操作函数，但需要注意有的参数类型不是 `ngx_str_t`，而是 `u_char*`：

```
#define ngx_strcmp(s1, s2)                //大小写敏感比较，参数是 u_char*
#define ngx_strncmp(s1, s2, n)            //大小写敏感比较，有长度参数

#define ngx_strstr(s1, s2)                //查找子串
#define ngx_strlen(s)                     //使用 '\0' 计算字符串长度

//大小写不敏感字符串比较，参数是 u_char*
ngx_int_t ngx_strcasecmp(u_char *s1, u_char *s2);
ngx_int_t ngx_strncasecmp(u_char *s1, u_char *s2, size_t n);
```

① 有些简单操作如 `ngx_strcmp`、`ngx_strstr`、`ngx_strlen` 等直接调用了 C 标准函数。


```
//字符串转整数类型, 参数是 u_char*
ngx_int_t ngx_atoi(u_char *line, size_t n);

//内存池复制字符串, 参数是 ngx_str_t*
u_char *ngx_pstrdup(ngx_pool_t *pool, ngx_str_t *src);
```

格式化函数

Nginx 实现了自己的类 printf() 格式化函数, 同样参数是 u_char*:

```
u_char * ngx_sprintf(u_char *buf, const char *fmt, ...);
u_char * ngx_snprintf(u_char *buf, size_t max, const char *fmt, ...);
u_char * ngx_slprintf(u_char *buf, u_char *last, const char *fmt, ...);
```

ngx_sprintf() 直接向 buf 输出格式化内容, 不检查缓冲区的有效性, 存在缓冲区溢出危险, 通常不建议使用。后两个函数比较安全, 参数 max 和 last 指明了缓冲区的结束位置, 所以格式化的结果只会填满缓冲区为止。

函数执行后会返回一个 u_char* 指针, 指示格式化输出后在 buf 里的结束位置, 可以用这个返回值来判断结果的长度。

Nginx 还定义了很多自己的格式化标志, 详细信息请参看附录 D。需要当心的是格式化输出 ngx_str_t 对象必须使用专有的 “%V”, 而不能是 “%s”。

3.5.3 C++封装

我们仍然利用 NgxWrapper 来封装 ngx_str_t 结构, 实现类 NgxString。读者可以在此基础上参考其他 Nginx 字符串函数实现更多的功能, 本书不再赘述。

类定义

NgxString 与 NgxPool 的实现相近, 定义如下:

```
class NgxString final : public NgxWrapper<ngx_str_t>
{
public:
    typedef NgxWrapper<ngx_str_t>    super_type;    //基本包装类
    typedef NgxString                this_type;     //简化类型定义

    typedef boost::string_ref        string_ref_type; //char*字符串引用类型
public:
    NgxString(ngx_str_t& str): super_type(str)        //构造
    {}
```

```

~NgxString() = default; //析构

... //成员函数见后

};

```

基本操作函数

NgxString 的接口模仿了标准字符串 `std::string`:

```

public:
    const char* data() const //获取字符串
    {
        return reinterpret_cast<const char*>(get()->data);
    }

    std::size_t size() const //获取长度
    {
        return get()->len;
    }

    bool empty() const //是否是空字符串
    {
        return !get()->data || !get()->len;
    }

```

这些成员函数都非常简单，无须过多说明。

为了让 NgxString 用起来更像 `std::string`，我们实现了一个函数 `str()`，它返回 `boost::string_ref` 类型，相当于一个标准常量字符串：

```

string_ref_type str() const //转换为 char* 字符串形式
{
    return string_ref_type(data(), size());
}

```

我们也可以编写 `begin()`、`end()`、`front()`、`back()` 等接口，这样它就可以直接搭配标准算法使用。限于篇幅这里不再列出详细实现源码，读者可以参考 GitHub 资源。

特殊操作函数

基于 Nginx 特有的字符串操作函数，还可以给 NgxString 添加其他有用的接口：^①

① `printf()` 没有调用成员函数 `data()`，这是因为 `data()` 使用了 `reinterpret_cast` 强制转型为了 `const char*`，而 Nginx 的字符串函数需要的是 `u_char*`。

```

public:
    operator ngx_int_t () const                //字符串转整数类型
    {
        return ngx_atoi(get()->data, get()->len);
    }

    //重载比较操作符, 大小写敏感比较两个 ngx_str_t 对象
    friend bool operator==(const this_type& l, const this_type& r)
    {
        return l.size() == r.size() &&
            ngx_strncmp(l.data(), r.data(), l.size()) == 0;
    }

    template<typename ... Args>                //可变模板参数列表
    void printf(const Args& ... args) const     //安全格式化
    {
        auto p = ngx_snprintf(get()->data, get()->len, args ...);

        get()->len =                          //计算实际长度
            static_cast<std::size_t>(p - get()->data);
    }

```

出于方便调试的目的, 本书还为 NgxString 重载了 C++ 流输出操作符:

```

public:
    template<typename T>
    friend T& operator<<(T& o, const this_type& s) //重载流输出操作符
    {
        o.write(s.data(), s.size());            //把字符串写入流
        return o;
    }

```

内存池复制字符串

ngx_str_t 的 ngx_pstrdup() 函数使用了内存池, 所以应该在类 NgxPool 里封装:

```

class NgxPool final : public NgxWrapper<ngx_pool_t>
{
public:
    ...
    ngx_str_t dup(ngx_str_t& str) const        //复制字符串
    {
        ngx_str_t tmp;                        //准备返回的字符串

        tmp.len = str.len;                    //设置字符串长度
        tmp.data = ngx_pstrdup(get(), &str);  //内存池复制字符串
    }

```

```

        NgxException::require(tmp.data);           //检查空指针
        return tmp;                                //返回复制的字符串
    }

    ngx_str_t dup(boost::string_ref str) const      //复制普通字符串
    {
        ngx_str_t tmp{                             //转换为 ngx_str_t 对象
            str.size(), (u_char*)str.data()};        //注意转型
        return dup(tmp);                             //调用 dup()
    }
};

```

3.6 时间与日期

与字符串一样，时间与日期是程序经常要处理的对象，在 Web 服务器里更是如此，Nginx 基于 C API 实现了比较全面的时间与日期功能。^①

3.6.1 时间结构定义

Nginx 定义了专用的时间数据结构 `ngx_time_t`：

```

// 定义在 core/ngx_times.h
typedef struct {
    time_t      sec;                //自 epoch 以来的秒数，即时间戳
    ngx_uint_t  msec;              //秒数后的小数部分，单位是毫秒
    ngx_int_t   gmtoff;            //GMT 时区偏移量
} ngx_time_t;

```

`ngx_time_t` 在 Nginx 里用来表示时间，成员变量 `sec` 和 `msec` 分别表示秒数（时间戳）和小数部分的毫秒数。`gmtoff` 是本地时区相对于 UTC/GMT 的偏移量，以分钟为单位，例如北京时区是 GMT+8，那么 `gmtoff` 就是 $480=8*60$ 。

3.6.2 时间操作函数

为了节约资源，避免频繁的系统调用，Nginx 内部使用了比较巧妙的 cache 机制来存放时间值，使用一个全局指针 `ngx_cached_time` 指示当前缓存的时间：

```
volatile ngx_time_t*  ngx_cached_time;    //当前缓存的时间
```

^① 如果觉得 Nginx 自身的时间功能不够，那么还可以使用 `boost.chrono` 和 `boost.date_time` 库。

两个函数宏 `ngx_time()` 和 `ngx_timeofday()` 可以分别获取当前时间的秒数（时间戳）和完整的时间数据结构：

```
#define ngx_time()          ngx_cached_time->sec
#define ngx_timeofday()    (ngx_time_t *) ngx_cached_time
```

基于 `ngx_cached_time`, Nginx 还定义了另外一个全局变量 `ngx_current_msec`, 表示自 epoch 以来的毫秒数, 即 `sec * 1000 + msec`:

```
volatile ngx_msec_t  ngx_current_msec;    //毫秒精度的时间戳
```

由于 `ngx_cached_time` 是一个缓存的时间, 可能存在延迟, 有时我们必须获取当前的精确时间, 这个时候可以调用 `ngx_time_update()` 强制更新缓存, 然后再获取时间:

```
void ngx_time_update(void);              //要求 Nginx 更新缓存的时间
```

但 `ngx_time_update()` 需要使用锁, 成本较高, 所以不应该频繁调用。

Nginx 还使用宏包装了系统调用 `sleep()`, 但它会阻塞整个进程, 应当慎用:

```
// 定义在 os/unix/nginx_time.h
#define ngx_msleep(ms)      (void) usleep(ms * 1000)
#define ngx_sleep(s)        (void) sleep(s)
```

3.6.3 日期结构定义

`ngx_time_t` 只表示计算机世界里的时间, 如果要使用现实生活里的年月日还需要另外一个结构 `ngx_tm_t`:

```
// 定义在 os/unix/nginx_time.h
typedef struct tm          ngx_tm_t;
```

`ngx_tm_t` 实际上是一个简单的 typedef, 它是标准 C 结构 `tm` 的同义词, `tm` 的定义是:

```
struct tm                // 定义在<ctime>
{
    int tm_sec;          // Seconds. [0-60] (1 leap second)
    int tm_min;          // Minutes. [0-59]
    int tm_hour;         // Hours. [0-23]
    int tm_mday;         // Day. [1-31]
    int tm_mon;          // Month. [0-11]
    int tm_year;         // Year 1900.
    int tm_wday;         // Day of week. [0-6]
    int tm_yday;         // Days in year. [0-365]
    int tm_isdst;        // DST. [-1/0/1]
};
```

为了让 struct tm 更“像”是 Nginx 自己的数据结构，Nginx 使用宏重命名了 tm 的成员变量，例如：

```
#define ngx_tm_mday      tm_mday      //天
#define ngx_tm_mon       tm_mon       //月
#define ngx_tm_year      tm_year      //年
#define ngx_tm_wday      tm_wday      //星期
```

3.6.4 日期操作函数

Nginx 提供两个函数，把 time_t 分别转换为格林尼治标准时间（GMT）和本地时间，time_t 值可以用 ngx_time() 得到：

```
// 定义在 core/nginx_times.h
void ngx_gmtime(time_t t, ngx_tm_t *tp);

// 定义在 os/unix/nginx_time.h
void ngx_localtime(time_t s, ngx_tm_t *tm);
```

Nginx 里还有几个函数能够在字符串和 time_t 之间互相转换处理，可以在处理 TCP/HTTP 请求时使用。

下面两个函数内部调用了 ngx_gmtime() 和 ngx_sprintf()，把 time_t 转换为日期字符串：

```
// 定义在 core/nginx_times.h
u_char *ngx_http_time(u_char *buf, time_t t);
u_char *ngx_http_cookie_time(u_char *buf, time_t t);
```

下面的函数解析字符串形式的日期时间，转换为 time_t：

```
// 定义在 core/nginx_parse_time.h
time_t ngx_parse_http_time(u_char *value, size_t len);
```

同样的，Nginx 使用全局变量提供缓存好的日期字符串，减少频繁调用的成本：

```
ngx_str_t  ngx_cached_err_log_time;      //错误日志的日期字符串
ngx_str_t  ngx_cached_http_time;        //HTTP 格式的日期字符串
ngx_str_t  ngx_cached_http_log_time;     //HTTP 日志的日期字符串
ngx_str_t  ngx_cached_http_log_iso8601; //ISO8601 格式的日期字符串
ngx_str_t  ngx_cached_syslog_time;       //系统日志格式的日期字符串
```

读者可以根据实际情况选择合适的日期字符串应用在自己的程序里。

3.6.5 C++封装时间

我们使用类 `NgxClock` 来封装 Nginx 里与 `ngx_time_t` 相关的操作^①, 实现一个简单的计时器。

类定义

`NgxClock` 的定义如下:

```
class NgxClock final //Nginx 的时间功能封装
{
public:
    NgxClock() = default;
    ~NgxClock() = default;
public:
    ... //成员函数见后
private:
    ngx_time_t m_time = now(); //初始化为当前时间
};
```

`NgxClock` 使用一个成员变量 `m_time` 来保存时间值, 在构造时调用成员函数 `now()` 获取当前时间, 表示计时的起点。

成员函数

静态成员函数 `now()` 首先调用 `ngx_time_update()`, 然后获取更新后的缓存时间:

```
public:
    static const ngx_time_t& now() //获取当前时间
    {
        ngx_time_update(); //更新缓存时间

        return *ngx_timeofday(); //获取缓存时间
    }
```

在创建了 `NgxClock` 对象后, 可以再次获取时间, 与保存的 `m_time` 之差就是流逝的时间:

```
public:
    ngx_time_t delta() const //计算流逝的时间
    {
        auto t = now(); //获取当前时间

        t.sec -= m_time.sec; //计算差值
        t.msec -= m_time.msec;
```

① 这里之所以使用 `Clock` 是因为 Nginx 内部还有定时器功能, `Timer` 的名字给定时器更合适。

```

        return t;
    }

    double elapsed() const                //返回浮点数格式的时间
    {
        auto t = delta();                //计算流逝的时间

        return t.sec + t.msec * 1.0 / 1000;    //转换为 double
    }

```

复位计时器操作也很有用，它能够重置计时起点：

```

public:
    void reset()                          //复位计时器，非 const
    {
        m_time = now();                  //重新获取当前时间
    }

```

NgxClock 也可以对 ngx_sleep() 等进行封装，这里不再列出代码，读者可参考 GitHub 资源。

用法

可以随时调用 NgxClock 的静态成员函数 now() 获取当前时间：

```

auto t = NgxClock::now();                //获取当前时间

```

当作计时器来使用时需要构造一个 NgxClock 对象开始计时，然后当操作结束时调用 elapsed() 得到计时结果：

```

NgxClock clock;                          //构造对象，开始计时
...                                       //执行某些操作
cout << clock.elapsed() << "s"<< endl;    //输出计时结果

```

3.6.6 C++封装日期

类 NgxDatetime 用来封装 Nginx 里日期相关的操作。

类定义

NgxDatetime 的定义如下：

```

class NgxDatetime final                  //Nginx 的日期功能封装
{
public:
    NgxDatetime() = default;

```



```

    ~NgxDatetime() = default;
public:
    ...                                     //成员函数见后
};

```

获取当天日期

获取当天日期是一个很常见的功能，实现也不难：调用 `ngx_time()` 得到时间戳，再转换为 `ngx_tm_t` 结构，就可以从它的成员得到年月日等值。

简单起见，我们只使用 `ngx_snprintf()` 生成 “yyyy-mm-dd” 格式的字符串，以 `ngx_str_t` 的形式输出：

```

public:
    static std::time_t since()                //封装 ngx_time()
    {
        return ngx_time();                  //返回时间戳
    }

    static ngx_str_t today()                  //获取当天日期
    {
        ngx_tm_t tm;                        //ngx_tm_t 对象

        ngx_localtime(since(), &tm);        //转换为本地时间

        static u_char buf[20] = {};         //静态字符数组用于输出

        auto p = ngx_snprintf(buf, 20,      //安全格式化，获取字符串末尾
                               "%d-%02d-%02d", //格式是 “yyyy-mm-dd”
                               tm.ngx_tm_year, tm.ngx_tm_mon, tm.ngx_tm_mday);

        return ngx_str_t{                   //返回 ngx_str_t 结构
            static_cast<std::size_t>(p - buf), buf};
    }

```

因为日期字符串很短，所以我们用一个很小的静态数组来保存格式化的字符串。`ngx_snprintf()` 的返回值指示了格式化结果的结束位置，可以计算得到字符串的长度生成 `ngx_str_t`。^①

如果需要其他格式的日期字符串，那么只要修改 `ngx_snprintf()` 里的格式字符串就可

① `today()` 函数可以进一步优化，采用与 Nginx 类似的缓存机制，在一天的时间里只执行一次 `ngx_snprintf()`，读者可以试试。

以了,但必须注意字符数组的大小,避免缓冲区溢出错误。

转换 http 字符串

NgxDatetime 使用两个重载的 `http()` 函数实现字符串与时间戳的相互转换,代码比较简单,基本是直接调用 Nginx 的接口函数:

```
public:
    static ngx_str_t http(std::time_t t = since()) //时间戳转日期字符串
    {
        static u_char buf[50] = {}; //静态字符数组用于输出

        auto p = ngx_http_time(buf, t); //转换为字符串

        return ngx_str_t{static_cast<std::size_t>(p - buf), buf};
    }

    static std::time_t http(ngx_str_t& str) //日期字符串转时间戳
    {
        return ngx_parse_http_time(str.data, str.len);
    }
```

用法

NgxString 里已经重载了 C++ 流输出操作符,所以可以直接流输出 `today()` 和 `http()` 函数:

```
cout << NgxDatetime::today() << endl; //输出今天日期
cout << NgxDatetime::http() << endl; //输出 http 格式 gmt 字符串
```

输出的结果是:

```
2017-02-17
Fri, 17 Feb 2017 06:58:12 GMT
```

3.7 运行日志

我们在 1.3.3 节初步了解了 Nginx 运行日志的配置^①,现在来学习如何在我们自己的代码里记录运行日志。

① 再强调一下,本书讨论的日志是 Nginx 内部的运行错误日志,实现在 `ngx_errlog_module` 中,而不是记录 TCP/HTTP 请求的访问日志。

3.7.1 结构定义

Nginx 使用结构体 ngx_log_t 表示运行日志，定义如下：

```
// 定义在 core/nginx_core.h
typedef struct ngx_log_s          ngx_log_t;          //简化的类型定义

// 定义在 core/nginx_log.h
struct ngx_log_s {                                //结构体定义
    ...                                           //暂不需要关心的内部成员
    ngx_uint_t  log_level;                        //日志级别
    ngx_log_t*  next;                             //日志对象链表指针
};
```

ngx_log_t 有很多内部成员，可以实现非常灵活的日志功能，但就我们目前开发的 TCP/HTTP 功能来说并不需要额外的定制工作，直接使用 ngx_log_t 对象就可以了。

3.7.2 操作函数

函数 ngx_log_error_core() 用来记录日志，它支持可变参数：①

```
// 定义在 core/nginx_log.h
void ngx_log_error_core(
    ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
    const char *fmt, ...);
```

ngx_log_error_core() 使用 ngx_log_t 对象记录 level 级别的日志，字符串消息的语法与 ngx_sprintf() 相同。

日志参数

日志级别参数 level 取值为下列的宏，它们对应配置文件里的 “debug|info|notice|warn|error|crit|alert|emerg” 级别：②

```
#define NGX_LOG_STDERR      0          //最高级别
#define NGX_LOG_EMERG      1
#define NGX_LOG_ALERT      2
#define NGX_LOG_CRIT       3
#define NGX_LOG_ERR        4          //常用级别
```

① Nginx 还提供其他日志工具，例如类似 printf() 的 ngx_log_stderr()、在配置时记录日志的 ngx_conf_log_error() 和专门记录调试日志的 ngx_log_debug()，本书暂不讨论，读者可自行研究。

② Nginx 的日志级别和 UNIX/Linux 的 syslog 是一致的，可以完全对应。

```

#define NGX_LOG_WARN          5           //常用级别
#define NGX_LOG_NOTICE        6
#define NGX_LOG_INFO           7
#define NGX_LOG_DEBUG          8           //最低级别

```

NGX_LOG_STDERR 是比 emerg 还要高的错误级别, 如果使用这个级别来记录日志, 那么 Nginx 将直接输出日志信息到标准错误输出 (通常是终端屏幕), 而不是写入日志文件。

常用的日志级别是 NGX_LOG_ERR 和 NGX_LOG_WARN。

err 参数表示系统调用失败返回的错误码, 即 errno:

```

// 定义在 os/unix/nginx_errno.h
typedef int          ngx_err_t;

```

因为我们只调用 Nginx 接口, 所以通常不会用到 err 参数, 可以把它简单地置为 0。

日志宏

函数 ngx_log_error_core() 只是记录日志, 它并不检查日志级别决定是否记录日志, Nginx 以宏的方式提供这个功能:

```

#define ngx_log_error(level, log, ...) \
    if ((log)->log_level >= level) \
        ngx_log_error_core(level, log, __VA_ARGS__)

```

可以看到, 只有当消息的日志级别高于 log 对象级别 (即消息的 level 值小) 时才会调用 ngx_log_error_core() 记录日志。

实际开发过程中应该使用宏 ngx_log_error 来记录日志, 尽量避免直接调用函数 ngx_log_error_core()。

3.7.3 C++封装

我们使用模板类 NgxLog 来封装 ngx_log_t 和日志操作。

类定义

NgxLog 的定义如下:

```

template<ngx_uint_t level = NGX_LOG_DEBUG>
class NgxLog final : public NgxWrapper<ngx_log_t>
{
public:
    typedef NgxWrapper<ngx_log_t>    super_type;    //基本包装类

```

```

typedef NgxLog                                this_type;    //简化类型定义
...                                           //成员函数见后
};

```

NgxLog 的模板参数比较特殊，使用了日志级别 level，这样做是为了简化调用接口，可以用 typedef 定义出不同日志级别的 log 类型：

```

typedef NgxLog<NGX_LOG_DEBUG>    NgxLogDebug;    //debug 级别
typedef NgxLog<NGX_LOG_INFO>     NgxLogInfo;     //info 级别
typedef NgxLog<NGX_LOG_WARN>     NgxLogWarning;   //warn 级别
typedef NgxLog<NGX_LOG_ERR>      NgxLogError;    //err 级别

```

构造函数

很多 Nginx 数据结构都有名为 log 的成员用于记录日志，所以 NgxLog 实现了模板构造函数：

```

public:
    NgxLog(ngx_log_t* l): super_type(l)           //直接使用 log 指针构造
    {}

    template<typename T>
    NgxLog(T* x) : super_type(x->log)             //使用 log 成员构造
    {}

```

但在处理 TCP/HTTP 请求时经常使用的 ngx_session_t/ngx_http_request_t 结构里却并没有直接的 log 成员，而是间接地通过 connection 成员持有，所以为了方便使用需要再实现一个重载构造函数：

```

template<typename T>                               //使用了模板元编程技术
NgxLog(T* x,...):                                  //connection 成员构造
    NgxLog(x->connection)                           //委托构造
{}

```

记录日志

成员函数 print() 封装了日志宏 ngx_log_error：

```

public:
    template<typename ... Args>                    //可变参数模板
    void print(const char* fmt, const Args& ... args) const
    {
        ngx_log_error(level, get(), 0, fmt, args...);
    }

    template<typename ... Args>                    //可变参数模板
    void print(ngx_err_t err, const Args& ... args) const

```

```
{
    ngx_log_error(level, get(), err, args...);
}
```

print() 有两种形式，第一种形式置 err 为 0，第二种形式允许用户传入特定的 errno。

用法

只要 Nginx 结构直接或者间接拥有 ngx_log_t 成员，我们就可以构造出 NgxLog 对象，调用它的 print() 记录运行日志，例如：

```
//假设已经有 ngx_http_request_t 对象 r
NgxLogError log(r);                                //使用别名类创建 NgxLog 对象
log.print("hello c++");                               //记录一条运行日志
```

这种方式适合需要记录大量日志的场合，log 对象可以重复使用。如果只是临时性地记录日志，那么也可以利用临时对象，在一条语句里完成对象的创建和日志记录操作：

```
NgxLogError(r).print("hello c++");                  //构造临时对象同时写入运行日志
```

在 error.log 里的记录是：

```
2017/xx/xx 12:05:22 [error] 4745#0: *1 hello c++, ...
```

3.8 总结

“勿在浮沙筑高塔”，在 Nginx 上做 C/C++ 的二次开发，必须要从底层开始学习，所以本章讨论的都是最基本、最常用的数据结构，包括整数类型、内存池、字符串、时间日期和错误日志，它们是 Nginx 宫殿的基石。

我们先研究了 Nginx 使用的整数类型。Nginx 定义了自己的一套整数类型，保证了跨平台的编译和运行。为了解决值未初始化的问题，Nginx “发明”了 UNSET 概念，用来确定值是否有效。

内存池是 Nginx 里非常重要的数据结构，它消灭了内存碎片和泄漏，提高了程序的运行效率，几乎所有的 Nginx 数据结构都会用到内存池，我们会在之后的章节里多次与它打交道，必须很好地掌握它的设计思想和用法。

字符串是 TCP/HTTP 处理时常用的对象，Nginx 使用引用的方式降低了内存拷贝的成本，可以高效地操作字符串。但使用时需要时刻注意 ngx_str_t 不同于我们之前的字符串概念，它是“只读”的，不能随意更改它的内容，如果确有必要可以使用内存池做一份拷贝再修改。

时间和日期也是 TCP/HTTP 处理时常见的对象，Nginx 为此提供了比较全面的操作，可

以高效地获取当前时间和日期，也可以转换成各种格式的字符串。

最后本章介绍了如何在 Nginx 里记录日志，它是检查运行状况和排查错误的必备工具。

Nginx 使用 C 语言实现了整个系统，但其中却蕴含着面向对象的思想，所以我们使用包装外观模式以 C++ 实现了这些数据结构对应的包装类，集成了数据结构和对应的操作，增强了类型的内聚性，同时没有效率的损失。我们还充分利用了 C++11 的特性，例如异常、类型自动推导、委托构造、可变参数模板等，让类的实现和接口更干净清楚，易用性和可维护性都比原始的 C 接口更好。读者可以从这些代码里仔细体会 C++ 的现代编程风格和思想。

第 4 章

Nginx高级数据结构

C 语言是一种小而简单的计算机语言，不仅语法简单，而且标准库也很简单，缺少实际开发工作中常用的数据结构和算法。这既是缺点也是优点，缺点是程序员必须自己实现所需的数据结构，优点则是给予了程序员最大的自由，允许他们充分发挥聪明才智，创造出各种各样令人惊讶不已的艺术品。

Nginx 就是 C 语言这一优点的杰出体现，它灵活运用了内存池和指针，针对 Web 服务器应用场景设计了数个高效的数据结构，包括动态数组、单向链表、双端队列等。Nginx 使用它们高效地处理各种数据，在源码里随处可见这些数据结构的身影。

本章将详细剖析 `ngx_array_t`、`ngx_list_t`、`ngx_queue_t`、`ngx_rbtrees_t` 等高级数据结构，并封装为更容易使用的 C++ 标准容器风格。

4.1 动态数组

C 语言的内置数组十分简陋，只为原始内存数据增加了一层薄薄的抽象，无法动态增长，使用上存在着诸多限制。

Nginx 借鉴了 C++ 标准容器 `std::vector`，以 C 语言实现了“泛型的”、可以在运行时随意变更大小的动态数组 `ngx_array_t`，而且由于使用 `ngx_pool_t` 分配内存保证没有内存泄漏。

4.1.1 结构定义

`ngx_array_t` 表示一块连续的内存，其中顺序存放着数组元素，概念上和原始数组很接近，它的定义是：


```
// 定义在 core/nginx_array.h
typedef struct {
    void*      elts;           //数组的内存位置，即数组首地址
    ngx_uint_t nelts;         //数组当前的元素数量
    size_t     size;          //数组元素的大小
    ngx_uint_t nalloc;        //数组可容纳的最多元素数量
    ngx_pool_t* pool;         //数组使用的内存池
} ngx_array_t;              //动态数组类型定义
```

ngx_array_t 里的成员 elts 就是 C 语言的原始数组，定义为 void* 指针，所以是“泛型”的，在实际使用时需要强制转换为真正的类型。

elts 之外的四个成员是动态数组的“元信息”，Nginx 利用这些信息来管理数组，可以对比标准容器 std::vector 来理解：

- nelts : 数组内元素的数量，相当于 vector.size();
- size : 元素的大小，相当于 sizeof<T>;
- nalloc : 数组的总容量，相当于 vector.capacity();
- pool : 数组使用的内存池，相当于 vector 的 allocator。

ngx_array_t 的内存布局如图 4-1 所示。

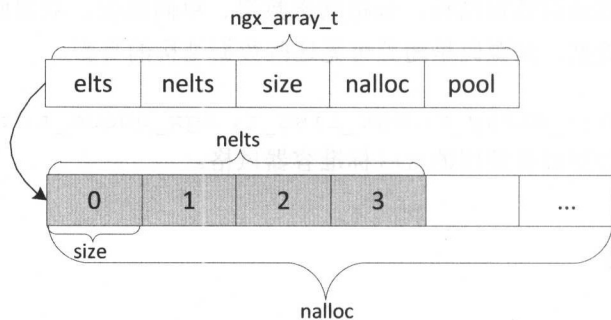


图 4-1 ngx_array_t 的内存布局

ngx_array_t 的 nalloc 虽然表示了总容量，但并不意味着只能容纳 nalloc 个元素。如果数组内的元素不断增加，当 nelts > nalloc 时将会引发数组扩容，ngx_array_t 会向内存池 pool 申请一块两倍原大小的空间——这个策略是和 std::vector 一样的。

但 ngx_array_t 的扩容成本较高，不仅要重新分配内存，而且要拷贝原有元素到新的位置，所以我们在使用 ngx_array_t 时最好一次性分配足够的空间，尽量避免动态扩容。

4.1.2 操作函数

`ngx_array_t` 的操作比较简单, 使用 `ngx_array_t.elts` 就可以访问数组里的元素, 不过之前必须要转换为实际的元素类型, 例如:

```
auto p = reinterpret_cast<T*>(arr.elts);           //类型转换
cout << p[0] << endl;                           //使用 operator[] 访问元素
```

在访问数组元素时需要注意越界问题, `ngx_array_t` 没有越界检查, 用户必须自行确保数组索引的有效性, 也就是说 $i < nelts$, 否则会导致未定义错误。

`ngx_array_t` 还提供创建、销毁和添加元素等常用的操作函数。

```
ngx_array_t* ngx_array_create(ngx_pool_t *p, ngx_uint_t n, size_t size);
void          ngx_array_destroy(ngx_array_t *a);
```

`ngx_array_create()` 函数使用内存池创建一个可容纳 n 个大小为 `size` 元素的数组, 即分配了一块 $n \times \text{size}$ 大小的内存块, `ngx_array_destroy()` 函数则“销毁”动态数组, 归还分配的内存。

```
void*          ngx_array_push(ngx_array_t *a);
void*          ngx_array_push_n(ngx_array_t *a, ngx_uint_t n);
```

这两个函数用来向数组添加元素, 用法比较特别, 它们返回的是一个 `void*` 指针, 用户必须把它转换为真正的元素类型再操作。不直接使用 `ngx_array_t.elts` 操作的原因是防止数组越界, 函数内部会检查当前数组容量自动扩容。

清空数组可以直接置 `nelts` 为 0, 但之前分配的内存并不会释放, 还可以用来存储数据。

4.1.3 C++动态数组

在 `NgxPool` 里可以封装 `ngx_array_create()` 函数, 创建动态数组:^①

```
class NgxPool final : public NgxWrapper<ngx_pool_t>
{
public:
    ...                                     //其他成员函数
    template<typename T>
    ngx_array_t* array(ngx_uint_t n = 10) const //默认容纳 10 个元素
    {
        auto p = ngx_array_create(get(), n, sizeof(T)); //使用内存池创建数组
    }
}
```

① `ngx_array_destroy()` 函数也可以同样封装, 但因为内存池会最终释放内存, 所以通常没有必要调用它。

```

        NgxException::require(p);                //检查空指针
    }
    return p;                                     //返回数组
};

```

然后我们实现 ngx_array_t 的 C++ 封装类 NgxArray。

类定义

NgxArray 使用 2.5 节的 NgxWrapper 代理 ngx_array_t，定义如下：

```

template<typename T>                                //T 是数组的元素类型
class NgxArray final : public NgxWrapper<ngx_array_t>
{
public:
    typedef NgxWrapper<ngx_array_t> super_type;      //简化类型定义
    typedef NgxArray this_type;
public:
    ...                                              //成员函数见后
};

```

字符串数组是 Nginx 里常见的数据结构，为了方便使用，可以定义一个特化类型：

```

typedef NgxArray<ngx_str_t> NgxStrArray;          //字符串数组类型

```

构造与析构

NgxArray 可以从内存池构造，或者从一个已经存在的 ngx_array_t 结构构造：

```

public:
    NgxArray(const NgxPool& p, ngx_uint_t n = 10):
        super_type(p.array<T>(n))                //调用内存池创建数组
    {}

    NgxArray(ngx_array_t* arr):super_type(arr)    //指针参数
    {}

    NgxArray(ngx_array_t& arr):super_type(arr)    //引用参数
    {}

    ~NgxArray() = default;                        //默认析构函数

```

访问元素

成员函数 elts() 获取 ngx_array_t.elts 指针，并转换为元素类型，注意它是私有的：

```

private:

```

```
T* elts() const //获取元素数组指针
{
    return reinterpret_cast<T*>(get()->elts); //类型转换
}
```

成员函数 `size()` 获取数组的大小，重载操作符 `[]` 来提供与原始数组一致的访问接口，并且还加入了越界检查，更加安全：

```
public:
    ngx_uint_t size() const //获得数组大小
    {
        return get()?get()->nelts:0; //防止空指针
    }

    T& operator[](ngx_uint_t i) const //重载操作符[]
    {
        NgxException::require(i < size() && get()); //越界、空指针检查

        return elts()[i]; //调用 elts() 访问元素
    }
```

添加元素

我们使用一个辅助函数 `prepare()` 封装 `ngx_array_push()`，它返回新数组元素的引用，然后 `push()` 就可以真正添加元素：^①

```
public:
    T& prepare() const
    {
        auto tmp = ngx_array_push(get()); //添加元素

        NgxException::require(tmp); //检查空指针

        return *reinterpret_cast<T*>(tmp); //返回元素的左引用
    }

    void push(const T& x) const
    {
        prepare() = x; //左引用可以直接赋值
    }
```

① 如果要追求效率，可以在 `push()` 里使用 `std::move()` 来消除拷贝的代价，当然我们也可以直接使用 `prepare()` 来操作数据。

其他操作

遍历数组元素是很常见的操作，使用 `for` 可以很容易实现，但一个泛型的遍历操作可能更有用，可以避免反复手写 `for` 循环。

成员函数 `visit()` 接受一个函数对象，对数组里的每一个元素执行该函数，类似标准算法 `for_each`：

```
public:
    template<typename V>
    void visit(V v) const           //访问数组里的所有元素
    {
        auto p = elts();            //获得数组首地址

        for(ngx_uint_t i = 0; i < size(); ++i)    //遍历数组
        {
            v(p[i]);                //执行函数对象
        }
    }
```

`NgxArray` 还可以有更多的功能，比如实现迭代器、符合标准容器定义、支持 `range-for` 等，读者可以根据需要参考后续章节自行实现。

用法

`NgxArray` 的接口很小，也很容易使用，如果读者熟悉 `std::vector` 或者 `boost::array`，那么就能很快掌握。

示范 `NgxArray` 用法的代码如下：

```
NgxArray<ngx_int_t> arr(pool);           //假设已有内存池，创建数组
assert(arr.size() == 0);                  //初始数组为空

arr.push(42);                             //添加元素
arr.push(253);
assert(arr[0] == 42);                     //访问元素

arr.visit(                                //遍历元素
    [](ngx_int_t x){                      //使用 lambda 表达式
        std::cout << x << ", ";
    });                                   //打印元素值，输出 42,253,
```

4.2 单向链表

链表也是一种常用的数据结构，与数组的连续内存不同，它使用指针来连接各个节点，所以会有额外的存储成本，但理论上来说可以存储任意数量的元素。

Nginx 的单向链表 `ngx_list_t` 设计融合了一些 `ngx_array_t` 的特点，在一个节点里存储多个元素，有效地降低了链表的存储成本。

4.2.1 结构定义

`ngx_list_part_t` 结构定义了链表的节点：

```
// 定义在 core/nginx_list.h
typedef struct ngx_list_part_s  ngx_list_part_t;

struct ngx_list_part_s {
    void*                elts;                // 数组元素指针
    ngx_uint_t           nelts;              // 数组里的元素数量
    ngx_list_part_t*     next;              // 下一个节点的指针
};
```

它类似 `ngx_array_t`，是一个简单的数组，也可以“泛型”存储数据，但少了一些成员，还有一个 `next` 指针指向链表里的下一个节点。

`ngx_list_t` 结构定义了链表，实际上是头节点+元信息：

```
// 定义在 core/nginx_list.h
typedef struct {
    ngx_list_part_t*  last;                // 链表的尾节点
    ngx_list_part_t   part;               // 链表的头节点
    size_t            size;               // 链表存储元素的大小
    ngx_uint_t        nalloc;            // 每个节点能够存储元素的数量
    ngx_pool_t*       pool;              // 链表使用的内存池
} ngx_list_t;
```

对比 `ngx_array_t` 可以看到，`ngx_list_t` 的成员 `size`、`nalloc` 和 `pool` 的含义是相同的，确定了节点里数组的元信息。可以这么说，链表里的每个节点就是一个简化的 `ngx_array_t` 数组结构。

`ngx_list_t` 的 `part` 成员是链表的头节点（注意不是指针），`part.next` 指向链表里的第二个节点，而 `last` 则指向链表的尾节点。

`ngx_list_t` 的内存布局如图 4-2 所示。

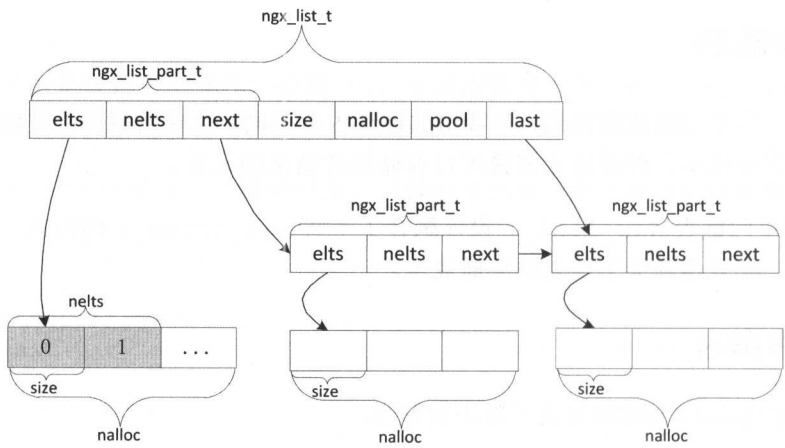


图 4-2 ngx_list_t 的内存布局

4.2.2 操作函数

函数 ngx_list_create() 使用内存池创建链表，接口与 ngx_array_create() 类似，每个节点可容纳 n 个大小为 size 的元素：

```
ngx_list_t *ngx_list_create(ngx_pool_t *pool, ngx_uint_t n, size_t size);
```

向链表里添加元素的操作 ngx_list_push() 也很类似，同样返回一个 void* 指针，需要转型操作：

```
void *ngx_list_push(ngx_list_t *list);
```

由于 ngx_list_t 是单向链表，所以我们不能使用索引来访问元素，必须从头节点开始用 next 指针逐个访问节点，这样在每个节点里才能用整数索引获取节点数组里的元素。

Nginx 在 <core/nginx_list.h> 里以注释的形式给出了访问列表元素的示例代码，如下：

```
part = &list.part; // 获取链表的头节点
data = part->elts; // 获得节点内数组地址

for (i = 0 ;; i++) { // 开始遍历链表

    if (i >= part->nelts) { // 检查是否节点数组越界
        if (part->next == NULL) { // 下一个节点指针
            break; // 指针为空表示链表结束
        }
    }

    part = part->next; // 跳到下一个节点
    data = part->elts; // 下一个节点的数组地址
}
```

```

        i = 0;                                //数组索引初始化
    }
    ... data[i] ...                            //在本节点内访问元素
}                                              //for 循环结束

```

4.2.3 C++迭代器

对于 ngx_list_t 这样的单向链表来说访问元素是非常重要的操作，在 C++里可以用迭代器的概念来表示，所以我们为 ngx_list_t 实现一个迭代器类 NgxListIterator。

早期实现完整的、符合标准的迭代器要花费很多力气，不过在 boost.iterator 库的 iterator_facade 工具类帮助下，这个工作就变得轻而易举了。^①

类定义

NgxListIterator 使用 iterator_facade 简化迭代器的定义：

```

template<typename T>
class NgxListIterator final :
    public boost::iterator_facade<                                //Boost 工具类
        NgxListIterator<T>, T,                                    //迭代器的值类型
        boost::single_pass_traversal_tag>                        //单向遍历
{
public:
    typedef boost::iterator_facade<...>      super_type;      //简化类型定义
    typedef typename super_type::reference    reference;
public:
    NgxListIterator(ngx_list_t* l):            //构造函数
        m_part(&l->part),                      //初始化头节点
        m_data(static_cast<T*>(m_part->elts))  //初始化数组指针
    {}

    NgxListIterator() = default;                //默认构造函数
    ~NgxListIterator() = default;               //默认析构函数
private:
    ngx_list_part_t* m_part = nullptr;         //节点指针
    T* m_data = nullptr;                       //节点内数组指针
    ngx_uint_t m_count = 0;                    //节点内数组索引
public:
    ...                                         //成员函数见后
};

```

^① boost::iterator_facade 的用法可参考推荐书目 [4]。

NgxListIterator 使用三个成员变量 `m_part`、`m_data` 和 `m_count` 来遍历链表，它们分别对应 Nginx 示例代码里的 `part`、`data` 和 `i`，构造函数里 `m_part` 和 `m_data` 指向头节点。

NgxListIterator 里的 `iterator_facade` 用法比较复杂，它使用模板参数在编译期帮助我们实现了迭代器的大部分操作和类型定义。如果不是特别熟悉迭代器概念和模板元编程可能会难以理解，不过这并不影响使用。

迭代器核心操作

`iterator_facade` 要求单向迭代器必须实现 `dereference()`、`increment()` 和 `equal()` 三个操作，然后利用这些函数实现 `operator++` 等迭代器接口：

```
private:
    friend class boost::iterator_core_access;           //必需的友元声明

    reference dereference() const                       //解引用操作
    {
        NgxException::require(m_data);                //检查空指针
        return m_data[m_count];                       //访问节点内数组元素
    }

    void increment()                                   //前进迭代器
    {
        if(!m_part || !m_data)                       //检查空指针
        { return; }

        ++m_count;                                     //仿造 Nginx 示例代码

        if(m_count >= m_part->nelts)                  //检查是否节点数组越界
        {
            m_count = 0;                               //数组索引初始化
            m_part = m_part->next;                     //跳到下一个节点

            m_data = m_part?                           //是否是尾节点?
                static_cast<T*>(m_part->elts):          //下一个节点的数组地址
                nullptr;                                //否则数组地址是空指针
        }
    }

    bool equal(NgxListIterator const& o) const         //比较两个迭代器
    {
        return m_part == o.m_part &&                 //必须是三个因素全相等
            m_data == o.m_data &&
```

```

        m_count == o.m_count;
    }

```

这里我们需要稍微注意 `increment()` 函数，它基本仿造了 Nginx 示例代码的 `for` 循环部分，但逻辑有小变化：因为迭代器遍历不会 `break`，所以当到达尾节点时迭代器的 `m_part` 等指针都会变成 0 表示结束，如果在这个已经结束的迭代器上继续前进可能发生未定义错误。

迭代器的有效性

为了防止迭代器结束时操作导致的错误，`NgxListIterator` 增加了 `bool` 转型操作符，用来检查空指针的情况：

```

public:
    BOOST_EXPLICIT_OPERATOR_BOOL()                //Boost 显式 bool 转型

    bool operator!() const                        //检查指针是否失效
    {
        return !m_part || !m_data || !m_part->nelts; //指针为空等情况则失效
    }

```

宏 `BOOST_EXPLICIT_OPERATOR_BOOL` 是 Boost 库提供的一个小工具，它利用 `operator!()` 实现安全的 `bool` 转型操作。^①

4.2.4 C++单向链表

`NgxList` 利用 `NgxWrapper` 封装了 `ngx_list_t`，创建链表的 `ngx_list_create()` 函数同样可以在 `NgxPool` 里封装，这里从略。

类定义

`NgxList` 的实现与 `NgxArray` 类似，定义如下：

```

template<typename T>                                //T 是数组的元素类型
class NgxList final : public NgxWrapper<ngx_list_t>
{
public:
    typedef NgxWrapper<ngx_list_t>    super_type;    //简化类型定义
    typedef NgxList                    this_type;

public:
    NgxList(ngx_list_t* l):super_type(l)            //指针参数

```

^① `BOOST_EXPLICIT_OPERATOR_BOOL` 的用法可参考推荐书目 [3]。

```

{}

NgxList(ngx_list_t& l):super_type(&l)           //引用参数
{}

~NgxList() = default;                           //默认析构函数
public:
    ...                                           //成员函数见后
};

```

添加元素

NgxList 同样使用函数 `prepare()` 和 `push()` 添加元素:

```

public:
    T& prepare() const
    {
        auto tmp = ngx_list_push(get());        //添加元素

        NgxException::require(tmp);              //检查空指针

        return *reinterpret_cast<T*>(tmp);        //返回元素的左引用
    }

    void push(const T& x) const
    {
        prepare() = x;                          //左引用可以直接赋值
    }

```

迭代器操作

NgxList 使用容器标准的 `begin()` 和 `end()` 操作产生迭代器:

```

public:
    typedef NgxListIterator<T>    iterator;      //迭代器类型定义
    typedef const iterator        const_iterator;

    iterator begin() const                       //迭代器起点
    {
        return iterator(get());                 //使用链表指针初始化
    }

    iterator end() const                         //迭代器终点
    {
        return iterator();                     //指针为空的迭代器, 逾尾
    }

```

使用迭代器可以很方便地遍历链表，例如实现元素的查找操作：

```
template<typename Predicate>
iterator find(Predicate p) const           //使用谓词函数对象查找
{
    auto iter = begin();                  //迭代器开始
    for(; iter; ++iter)                  //结束条件是迭代器无效
    {
        if(p(*iter))                    //谓词检查元素
        {
            return iter;                 //满足条件则返回迭代器
        }
    }

    return end();                         //未找到则返回逾尾迭代器
}
```

用法

经过 `NgxList` 的封装后，`Nginx` 的单向链表变得非常容易使用，很像标准容器 `std::forward_list`：

```
NgxList<ngx_int_t> l(r, 1);                //节点里的数组只保存一个元素

l.push(2000);                               //添加三个元素
l.push(2008);
l.push(2017);

assert(*l.begin() == 2000);                 //取第一个元素

auto p = l.find(                            //查找元素，返回迭代器
    [](ngx_int_t x){                        //lambda 表达式
        return x == 2017;                  //查找元素 2017
    });

assert(p != l.end());                       //必定查找到
assert(*p == 2017);

for(auto p = l.begin(); p++; p)            //遍历链表，也可以用范围 for
{
    std::cout << *p << ", ";              //输出 2000,2008,2017,
}
```

4.3 双端队列

双端队列是在两端都可以插入或删除元素的数据结构，在 Nginx 里它被实现为双向循环链表 `ngx_queue_t`，类似标准容器 `std::list`。

借用 Boost 程序库的概念，前面的 `ngx_array_t` 和 `ngx_list_t` 属于非侵入式容器，元素无须改动即可加入容器，而 `ngx_queue_t` 则不同，它是侵入式容器，必须把 `ngx_queue_t` 作为元素的一个成员，然后才能放入队列，与 `boost.intrusive` 库很接近。

4.3.1 结构定义

`ngx_queue_t` 的定义非常简单，不包含任何元素信息，只有链表的前后两个指针：

```
// 定义在 core/ngx_queue.h
typedef struct ngx_queue_s  ngx_queue_t;

struct ngx_queue_s {
    ngx_queue_t* prev;           //前驱指针
    ngx_queue_t* next;          //后继指针
};
```

`ngx_queue_t` 的使用方式类似 `boost.intrusive` 库的成员挂钩 (member hook)，结构体里需要添加它作为成员，例如：

```
struct Xinfo                      //一个可放入队列的数据结构
{
    int x = 0;                    //数据结构携带的信息
    ngx_queue_t queue;            //ngx_queue_t 成员，名字任意
};
```

通过这种方式，`ngx_queue_t` “侵入”了数据结构的内部，为数据结构增加了双向链表所需的两个链接指针。

`ngx_queue_t` 的内存布局如图 4-3 所示。

数据结构里可以有不止一个 `ngx_queue_t` 成员，这意味着它可以同时属于多个不同的双向链表（例如 `ngx_resolver_t` 结构体）。

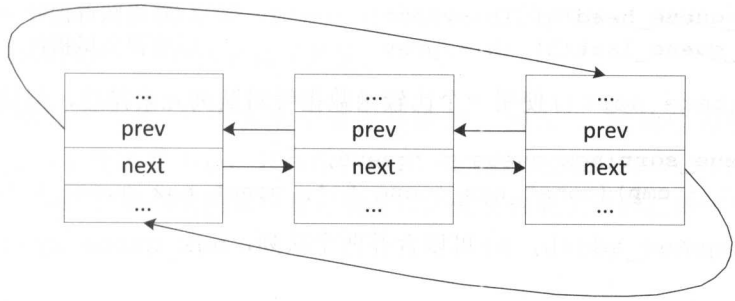


图 4-3 ngx_queue_t 的内存布局

4.3.2 操作函数

ngx_queue_t 使用一个头节点来表示队列，这个头节点可以是单纯的 ngx_queue_t 结构，不存储任何信息，但单从 ngx_queue_t 结构来看，它与普通的数据节点并无区别，我们只能根据实际情况区分。

Nginx 提供数个函数宏来调整 ngx_queue_t 指针操作队列，有的只对头节点有意义，参数的名字是 h (head)，但语法层面无法做到强制要求，这是它们的一个缺点。

头节点操作

函数宏 ngx_queue_init() 初始化头节点，把两个指针都指向自身：

```
#define ngx_queue_init(q) \
    (q)->prev = q; \
    (q)->next = q
```

函数宏 ngx_queue_sentinel() 返回节点自身，对于头节点来说就相当于哨兵：

```
#define ngx_queue_sentinel(h) (h)
```

ngx_queue_empty() 检查头节点的前驱指针，判断是否是空队列：

```
#define ngx_queue_empty(h) \
    (h == (h)->prev)
```

函数宏 ngx_queue_insert_head() 和 ngx_queue_insert_tail() 用来向队列的头尾插入数据节点：

```
#define ngx_queue_insert_head(h, x) \
#define ngx_queue_insert_tail(h, x)
```

函数宏 ngx_queue_head() 和 ngx_queue_last() 获取队列的头尾指针，可以用它们来实现队列的正向或反向遍历，直到遇到头节点 (ngx_queue_sentinel) 停止：

```
#define ngx_queue_head(h) (h)->next           //获得队首指针
#define ngx_queue_last(h) (h)->prev           //获得队尾指针
```

函数 `ngx_queue_sort()` 使用一个比较函数指针对队列元素排序，但效率不是很高：

```
void ngx_queue_sort(ngx_queue_t *queue,
    ngx_int_t (*cmp)(const ngx_queue_t *, const ngx_queue_t *));
```

此外，`ngx_queue_add(h, n)` 可以合并两个队列，`ngx_queue_split(h, q, n)` 可以拆分队列。

数据节点操作

数据节点本质上与头节点并无区别，所以很多操作代码是相同的。

函数宏 `ngx_queue_insert_after()` 在节点的后面插入数据，它其实就是 `ngx_queue_insert_head`：

```
#define ngx_queue_insert_after ngx_queue_insert_head
```

在节点前插入数据可以用宏 `ngx_queue_insert_tail()`，很可惜，它没有被重新定义为 `ngx_queue_insert_before`。

节点的前驱指针和后继指针可以使用函数宏 `ngx_queue_next()` 和 `ngx_queue_prev()` 获得，它们与 `ngx_queue_head()` 和 `ngx_queue_last()` 是一样的：

```
#define ngx_queue_next(q) (q)->next           //获得下一个节点的指针
#define ngx_queue_prev(q) (q)->prev           //获得前一个节点的指针
```

函数宏 `ngx_queue_remove()` 可以“删除”当前节点，实际上它只是调整了节点的指针，把节点从队列里摘除，并没有真正从内存里删除数据：

```
#define ngx_queue_remove(x) \
    (x)->next->prev = (x)->prev; \
    (x)->prev->next = (x)->next           //从队列里“删除”节点
```

Nginx 采用了一种巧妙（或者说有些 hack）的方法，可以从作为数据成员的 `ngx_queue_t` 结构访问到完整的数据节点：

```
#define ngx_queue_data(q, type, link) \
    (type *) ((u_char *) q - offsetof(type, link)) //获取节点数据 //返回结构体指针
```

函数宏 `ngx_queue_data()` 的参数分别是 `ngx_queue_t` 指针、节点类型和 `ngx_`

queue_t 成员名，它利用 C 标准宏 `offsetof` 计算成员在结构体里的偏移量^①，然后再倒着减去偏移量，这样就得到了结构体的真正地址。

这种获取数据的方法在 C 语言里是毫无问题的，因为 C 结构的内存布局是平坦的 (Plain)，但在 C++ 里则需要小心，只有符合 POD (Plain Old Data) 的结构体才能这样操作，否则 `offsetof` 计算得到的偏移量可能有误，无法获取正确的结构体地址。

结构分析

Nginx 的队列 `ngx_queue_t` 虽然结构很简单，但应用是比较复杂的，很多概念都混合在了一起，如果理解不透彻很容易误用。

通过上面的分析，我们可以把它们分解为节点、迭代器和队列容器三个概念：节点保存数据，迭代器遍历数据，而队列容器则是头节点。这三个概念可以使用 C++ 封装为不同的类，达到解耦的目的。

4.3.3 C++ 节点

封装队列节点必须要解决两个问题：

- 已知结构体 T，如何获取 T 内的 `ngx_queue_t` 成员；
- 已知结构体 T 和 `ngx_queue_t` 指针，如何获取 T 内的 `ngx_queue_t` 成员偏移量，进而得到对象地址。

Nginx 源码把这两个问题完全抛给了用户，要求用户自己手写结构体内的成员。不能说它“不负责”，这完全是因为 C 语言自身的能力限制。但在 C++ 里，使用强大的模板技术，再加上仔细的设计，我们完全可以抛开宏，在编译期就得到这些信息，从而简化客户代码。

类定义

`NgxQueueNode` 封装了 `ngx_queue_t` 的数据节点操作，它在模板参数列表里解决了刚才的两个问题，定义如下：

```
template<typename T,
        ngx_queue_t T::* q = &T::queue,           //成员变量指针类型
        std::size_t offset =                      //成员偏移量
            (std::size_t)&(((T*)0)->*q)           //模仿 offsetof 宏
>
```

① `offsetof` 是 C 标准里定义的一个宏，用来计算结构里成员的偏移量，相当于 `((size_t)&(((struct*)0)->member))`。


```

class NgxQueueNode final : public NgxWrapper<ngx_queue_t>
{
public:
    typedef NgxWrapper<ngx_queue_t> super_type;    //简化类型定义
    typedef NgxQueueNode this_type;

public:
    NgxQueueNode(ngx_queue_t* ptr): super_type(ptr) //直接指针构造
    {}

    NgxQueueNode(T& x): super_type(x.*q)           //获取成员变量地址
    {}

    NgxQueueNode(T* x): super_type(x->*q)          //获取成员变量地址
    {}

    ~NgxQueueNode() = default;                     //默认析构函数
public:
    ...                                           //成员函数见后
};

```

NgxQueueNode 使用了 C++ 里比较少用的特性——成员变量指针，形式是“type T::*”，如果有类型 T 的实例 x，那么可以用 x.*ptr 方式访问 x 的成员变量。所以，只要用户在模板参数里以“&T::member_name”的形式给出成员变量指针，那么就可以使用这个指针“进入”结构体内部得到 ngx_queue_t 的位置。

有了成员变量指针，我们再参考 offsetof 宏的实现原理，把 0 强制转换为 T* 类型，访问其成员，就能够计算出成员的偏移量（同样要求 T 是 POD 类型）。

操作函数

NgxQueueNode 获取节点数据的操作和宏 ngx_queue_data() 很相似，但它已经通过模板参数在编译期得到了数据节点 T 里与 ngx_queue_t 相关的信息，所以完全不需要用户的参与：

```

public:
    T& data() const                                //获取节点数据
    {
        return *(T*)((u_char*)(get()) - offset); //返回引用
    }

```

其他数据节点操作都很简单，直接调用 Nginx 的函数宏：

```

public:
    NgxQueueNode next() const                      //下一个节点
    {

```

```

    return ngx_queue_next(get());
}

NgxQueueNode prev() const                //前一个节点
{
    return ngx_queue_prev(get());
}

public:
    void append(T& v) const                //插入后继节点
    {
        ngx_queue_insert_after(get(), &(v.*q));
    }

    void remove() const                    //删除当前节点
    {
        ngx_queue_remove(get());
    }

```

需要注意 `ngx_queue_insert_after()` 里成员变量指针的调用方式，因为 `&` 操作符的优先级高，所以获取成员变量的表达式 “`v.*q`” 必须要用圆括号括起来。

4.3.4 C++迭代器

`ngx_queue_t` 是双向链表，用来遍历它的迭代器也必然是双向的，我们仍然使用 `iterator_facade` 来辅助定义迭代器，但迭代器的 `tag` 需要改用 `bidirectional_traversal_tag`。

类定义

`NgxQueueIterator` 封装了 `ngx_queue_t` 的数据节点操作，定义如下：

```

template<typename T>                                //T 是 NgxQueueNode
class NgxQueueIterator final :
    public boost::iterator_facade<
        NgxQueueIterator<T>, T,
        boost::bidirectional_traversal_tag>        //双向迭代器
{
public:
    typedef boost::iterator_facade<...> super_type;    //简化类型定义
    typedef NgxQueueIterator this_type;

    typedef typename super_type::reference reference;    //引用类型
public:
    NgxQueueIterator(ngx_queue_t* p): m_cur(p)        //初始化指针

```

```

{}

NgxQueueIterator() = default;           //默认构造函数
~NgxQueueIterator() = default;
public:
...                                     //成员函数见后
private:
    ngx_queue_t* m_cur = nullptr;       //迭代用的指针
    mutable T     m_proxy{m_cur};       //代理节点对象
};

```

为了方便使用, `NgxQueueIterator` 模仿了 C++ 的 `std::vector<bool>`, 内部构造了一个 `NgxQueueNode` 对象作为节点的代理, 在解引用时用户可以用这个代理对象直接访问真正的节点数据。

迭代器核心操作

双向迭代器除 `dereference()`、`increment()` 和 `equal()` 以外还需要提供 `decrement()`, 实现反向迭代功能:

```

private:
    friend class boost::iterator_core_access; //必需的友元声明

    reference dereference() const           //解引用操作, 返回代理对象
    {
        m_proxy = m_cur;                   //拷贝赋值数据节点
        return m_proxy;                     //返回节点对象的引用
    }

    void increment()                         //前进迭代器
    {
        m_cur = ngx_queue_next(m_cur);
    }

    void decrement()                         //后退迭代器
    {
        m_cur = ngx_queue_prev(m_cur);
    }

    bool equal(this_type const& o) const    //比较两个迭代器
    {
        return m_cur == o.m_cur;
    }

```

`NgxQueueIterator` 的解引用操作比较特殊, 它并不直接返回迭代器内部数据 (因为

`m_cur` 其实是 `ngx_queue_t`, 毫无意义), 而是返回 `NgxQueueNode` 对象作为代理。

4.3.5 C++双端队列

`NgxQueue` 封装了 `ngx_queue_t` 的队列操作, 它实际上代理了队列的头节点。

类定义

`NgxQueue` 的模板参数与 `NgxQueueNode` 相同, 也需要计算类型的成员变量指针和偏移量:

```
template<typename T,
        ngx_queue_t T::* q = &T::queue,           //成员变量指针类型
        std::size_t offset =                      //成员偏移量
            (std::size_t)&((T*)0)->*q)           //模仿 offsetof 宏
>
class NgxQueue final : public NgxWrapper<ngx_queue_t>
{
public:
    typedef NgxWrapper<ngx_queue_t>      super_type; //简化类型定义
    typedef NgxQueueNode<T, q, offset>   node_type;  //节点类型定义
    typedef NgxQueue                     this_type;

public:
    NgxQueue(ngx_queue_t& v): super_type(&v)        //引用构造
    {}

    NgxQueue(ngx_queue_t* ptr): super_type(ptr)      //指针构造
    {}

    NgxQueue(T& x): super_type(x.*q)                //获取成员变量地址
    {}

    NgxQueue(T* x): super_type(x->*q)                //获取成员变量地址
    {}

    ~NgxQueue() = default;                          //默认析构函数

public:
    ...                                              //成员函数见后
};
```

`NgxQueue` 的模板参数、构造函数与 `NgxQueueNode` 基本相同, 这并不奇怪, 因为它们代理的是 `ngx_queue_t`, 都表示队列里的节点, 只不过两者的作用不同: `NgxQueue` 是头节点, 而 `NgxQueueNode` 是数据节点。

基本操作

NgxQueue 的基本操作有初始化、判断是否空、取首尾节点等：

```
public:
    void init() const //初始化队列
    {
        ngx_queue_init(get());
    }

    bool empty() const //队列是否为空
    {
        return ngx_queue_empty(get());
    }
public:
    node_type front() const //获得队首节点
    {
        return ngx_queue_head(get());
    }

    node_type back() const //获得队尾节点
    {
        return ngx_queue_last(get());
    }
```

当队列为空时，`front()` 和 `back()` 会返回无效的 `NgxQueueNode` 对象，可以用 `operator bool` 来检查。

迭代器操作

NgxQueue 支持双向遍历，所以除 `begin()` 和 `end()` 函数外还有 `rbegin()` 和 `rend()` 函数，后两个函数返回逆向迭代器：

```
public:
    //迭代器和逆向迭代器类型定义
    typedef NgxQueueIterator<node_type> iterator;
    typedef boost::reverse_iterator<iterator> reverse_iterator;

    typedef const iterator const_iterator;
    typedef const reverse_iterator const_reverse_iterator;

    iterator begin() const //正向迭代器起点
    {
        return iterator(ngx_queue_head(get()));
    }
```

```

iterator end() const //正向迭代器终点
{
    return iterator(ngx_queue_sentinel(get())); //终点即头节点
}

reverse_iterator rbegin() const //逆向迭代器起点
{
    return reverse_iterator(end()); //适配正向迭代器
}

reverse_iterator rend() const //逆向迭代器终点
{
    return reverse_iterator(begin()); //适配正向迭代器
}

```

这里我们使用了 Boost 程序库的另一个迭代器工具 `reverse_iterator`，它可以把一个符合规范的正向迭代器适配成逆向迭代器。

添加元素

`NgxQueue` 的成员函数 `insert()` 和 `append()` 分别向队列两端添加节点：

```

public:
    void insert(T& v) const //插入队首
    {
        ngx_queue_insert_head(get(), &(v.*q));
    }

    void append(T& v) const //插入队尾
    {
        ngx_queue_insert_tail(get(), &(v.*q));
    }

```

使用迭代器表示的位置可以在队列中间插入节点：

```

void insert(const iterator& iter, T& v) const //插入迭代器位置之后
{
    iter->append(v);
}

```

其他操作

`NgxQueue` 还可以封装 `ngx_queue_add()`、`ngx_queue_sort()` 等函数，限于篇幅不再罗列，读者可参考 GitHub 资源。

用法

示范 NgxQueue 用法的代码如下:

```
typedef NgxQueue<Xinfo> queue_type;           //队列类型定义, 成员变量指针默认
ngx_queue_t h;                                //队列的头节点, 无数据

queue_type q(h);                              //代理队列头节点
q.init();                                     //初始化队列
assert(q.empty());                             //此时队列为空

Xinfo arr[3];                                  //创建三个数据节点
arr[0].x = 100;
arr[1].x = 200;
arr[2].x = 300;

q.insert(arr[0]);                             //添加到队列头
q.insert(arr[1]);                             //添加到队列头
q.append(arr[2]);                             //添加到队列尾
assert(q.front().data().x == 200);            //访问队首元素

for(auto& x : q)                               //范围 for 遍历队列
{
    cout << x.data().x << "-";               //输出 200-100-300-
}

auto last = q.back();                         //获取队尾元素
last.remove();                                //删除队尾元素

for(auto& x : q)                               //再次范围 for 遍历队列
{
    cout << x.data().x << "-";               //输出 200-100-
}

BOOST_REVERSE_FOREACH(auto& x, q)             //用 boost.foreach 反向遍历
{
    cout << x.data().x << ", ";              //输出 100,200,
}
```

使用 NgxQueue 必须定义好它的模板参数, 最重要的是 ngx_queue_t 成员变量指针, 如果成员的名字是默认的 queue 就可以省略, 否则必须以 `&T::member` 的形式给出。例如对于结构 ngx_resolver_s/ngx_resolver_t:

```
struct ngx_resolver_s {
    ...                                     //其他成员
```

```
    ngx_queue_t      name_resend_queue;    //ngx_queue_t 成员
};
```

NgxQueue 的模板参数就是：

```
typedef NgxQueue<
    ngx_resolver_t, &ngx_resolver_t::name_resend_queue> queue_type;
```

4.4 红黑树

红黑树是一种自平衡二叉树，不仅可以使使用二分法快速查找，而且插入和删除操作的效率也很高，常用于构造关联数组（例如 C++ 标准库里的 set 和 map）。

在 Nginx 里红黑树主要用在事件机制里的定时器，检查连接超时，此外还在 reslover、cache 里用于快速查找。

4.4.1 节点结构定义

Nginx 重定义了 ngx_uint_t 和 ngx_int_t 作为红黑树的键：

```
// 定义在 core/nginx_rbtrees.h
typedef ngx_uint_t  ngx_rbtrees_key_t;    //红黑树键类型，无符号整数
typedef ngx_int_t   ngx_rbtrees_key_int_t; //红黑树键类型，有符号整数
```

ngx_rbtrees_node_t 是红黑树的节点，它包含左右指针、父指针和颜色等信息：

```
// 定义在 core/nginx_rbtrees.h
typedef struct ngx_rbtrees_node_s  ngx_rbtrees_node_t;

struct ngx_rbtrees_node_s {
    ngx_rbtrees_key_t    key;                //红黑树的键，用于排序比较
    ngx_rbtrees_node_t*  left;              //左节点指针
    ngx_rbtrees_node_t*  right;             //右节点指针
    ngx_rbtrees_node_t*  parent;            //父节点指针
    u_char                color;             //节点的颜色，1 表示红色，0 表示黑色
    u_char                data;              //节点数据，只有一个字节，通常无意义
};
```

与 ngx_queue_t 一样，ngx_rbtrees_node_t 也要作为结构体的一个成员，以“侵入”的方式来使用。例如保存字符串的红黑树节点是：

```
typedef struct {
    ngx_rbtrees_node_t  node;                //红黑树节点，但不必是第一个成员
    ngx_str_t            str;                //节点的其他信息
} ngx_str_node_t;
```


4.4.2 树结构定义

`ngx_rbtrees_t` 定义了红黑树:

```
typedef struct ngx_rbtrees_s ngx_rbtrees_t;    //简化类型定义

//节点的插入方法, 函数指针类型
typedef void (*ngx_rbtrees_insert_pt) (ngx_rbtrees_node_t *root,
    ngx_rbtrees_node_t *node, ngx_rbtrees_node_t *sentinel);

struct ngx_rbtrees_s {
    ngx_rbtrees_node_t*      root;              //红黑树的根节点
    ngx_rbtrees_node_t*      sentinel;         //哨兵节点, 相当于空指针、空对象
    ngx_rbtrees_insert_pt    insert;           //节点的插入方法
};
```

`ngx_rbtrees_t` 有三个成员, 分别是:

- `root` : 红黑树的根节点;
- `sentinel` : 红黑树的哨兵节点, 用于简化实现, 它总是黑色的;
- `insert` : 节点的插入方法, 允许对特殊节点定制特殊插入方法。

红黑树结构体里的重要成员是函数指针 `insert`, 它决定了红黑树节点的实际插入操作, 用户可以针对不同的节点类型实现不同的插入方法, 但必须符合 `ngx_rbtrees_insert_pt` 的定义: 以 `root` 为根节点, 以 `sentinel` 为哨兵节点, 把节点 `node` 按照二叉树的规则插入到合适的位置。

`ngx_rbtrees_t` 的内存布局如图 4-4 所示。

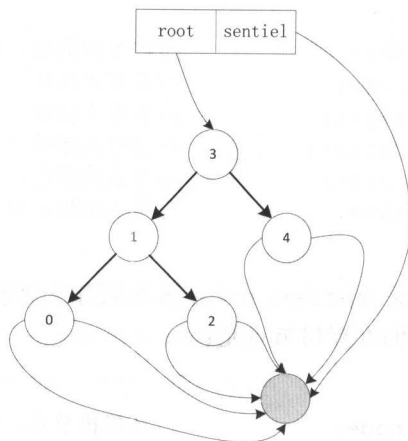


图 4-4 `ngx_rbtrees_t` 的内存布局

Nginx 提供了三个常用的插入函数：

//红黑树的键值是标准的 uint/int

```
void ngx_rbtrees_insert_value(
    ngx_rbtrees_node_t *root,
    ngx_rbtrees_node_t *node, ngx_rbtrees_node_t *sentinel);
```

//定时器红黑树的专用插入函数，键值是毫秒值

```
void ngx_rbtrees_insert_timer_value(
    ngx_rbtrees_node_t *root,
    ngx_rbtrees_node_t *node, ngx_rbtrees_node_t *sentinel);
```

//字符串红黑树的专用插入函数，键值是字符串的 hash 值

```
void ngx_str_rbtrees_insert_value(
    ngx_rbtrees_node_t *temp,
    ngx_rbtrees_node_t *node, ngx_rbtrees_node_t *sentinel);
```

参考这三个函数可以很容易地实现自己的插入函数，例如：

```
void ngx_rbtrees_insert_value(...) //插入标准的红黑树，键值是整数
{
    ngx_rbtrees_node_t **p; //树节点指针

    for ( ;; ) {
        p = (node->key < temp->key) ? //比较当前节点与插入节点
            &temp->left : &temp->right; //决定走左边还是右边

        if (*p == sentinel) { break; } //直到遇到哨兵节点结束
        temp = *p; //移动当前节点指针
    }

    *p = node; //此时已找到了合适的位置
    node->parent = temp; //设置插入节点的父节点
    node->left = sentinel; //左节点置为哨兵节点
    node->right = sentinel; //右节点置为哨兵节点
    ngx_rbt_red(node); //节点颜色是红色
}
```

4.4.3 操作函数

初始化红黑树需要指定哨兵节点和节点的插入函数：

```
#define ngx_rbtrees_init(tree, s, i) //tree 使用 s 作为哨兵节点，插入方法是 i
```

ngx_rbtrees_init 实际上是一个宏，初始化后红黑树里仅有一个哨兵节点 s，它同时也是根节点。

红黑树节点的添加和删除需要使用下面的两个函数，参数是红黑树结构体和树节点指针，如果操作后树的平衡性被破坏会自动旋转以保持平衡：

```
void ngx_rbtree_insert(ngx_rbtree_t *tree, ngx_rbtree_node_t *node);
void ngx_rbtree_delete(ngx_rbtree_t *tree, ngx_rbtree_node_t *node);
```

基本的红黑树只提供查找最小节点的功能，因为红黑树属于二叉树，所以查找速度非常快，只是从当前节点开始顺着指针找到最左边的节点：

```
ngx_rbtree_node_t* ngx_rbtree_min(
    ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel)
{
    while (node->left != sentinel) {           //哨兵节点是判断条件
        node = node->left;                     //遍历左子树
    }
    return node;                               //找到最左边的节点
}
```

ngx_rbtree_min() 只返回 ngx_rbtree_node_t* 指针类型，如果想得到完整的结构体指针，就要采用与 ngx_queue_t 同样的方法——利用 offsetof 宏计算偏移量再强制类型转换。

Nginx 还提供查找“下一个”节点的功能，利用它可以实现正序遍历红黑树：^①

```
ngx_rbtree_node_t *ngx_rbtree_next(
    ngx_rbtree_t *tree, ngx_rbtree_node_t *node);
```

对于常用的字符串红黑树，Nginx 提供了专用的查找函数，它可以在树里找到任意的字符串，如果字符串不存在则返回 nullptr：

```
ngx_str_node_t *ngx_str_rbtree_lookup(
    ngx_rbtree_t *rbtree, ngx_str_t *name, uint32_t hash);
```

当然，因为红黑树的结构是已知的，我们也可以自行编写代码去查找任意的键值，实现起来并不复杂，本书从略。

4.4.4 C++红黑树

类 NgxRbtree 使用与 NgxQueue 相同的成员变量指针技术封装 ngx_rbtree_t 结构，实现 Nginx 红黑树。

^① ngx_rbtree_next() 函数是 Nginx 1.11.11 新增的，之前的版本没有，请读者注意。

类定义

NgxRbtree 的定义如下:

```
template<typename T,                                //节点数据类型
        ngx_rbtree_node_t T::* np,                //红黑树节点成员函数指针
        ngx_rbtree_insert_pt func                  //插入方法函数指针
>
class NgxRbtree final : public NgxWrapper<ngx_rbtree_t>
{
public:
    typedef NgxWrapper<ngx_rbtree_t> super_type;    //简化类型定义
    typedef NgxRbtree this_type;

    typedef ngx_rbtree_key_t key_type;
    typedef ngx_rbtree_t tree_type;
    typedef ngx_rbtree_node_t node_type;

public:
    NgxRbtree(tree_type* t) : super_type(t)         //构造函数
    {}

    ~NgxRbtree() = default;                          //析构函数

public:
    ...                                              //成员函数见后
};
```

这里的模板参数 T 是一个含有 ngx_rbtree_node_t 成员的结构体, 因为使用了成员变量指针, 所以 ngx_rbtree_node_t 不必是第一个成员, 在结构体里的任意位置都可以。

基本操作

NgxRbtree 的基本操作有初始化、判断是否空、添加删除节点等:

```
public:
    static void init(tree_type& tree, node_type& s) //初始化一个红黑树
    {
        ngx_rbtree_init(&tree, &s, func);
    }

    bool empty() const //判断红黑树是否为空
    {
        return get()->root == get()->sentinel;
    }

public:
    void add(T& v) const //插入节点
    {
```

```

    ngx_rbtrees_insert(get(), &(v.*np));
}

void del(T& v) const //删除节点
{
    ngx_rbtrees_delete(get(), &(v.*np));
}

```

查找操作

基本的红黑树查找操作是查找最小节点，但只能返回 ngx_rbtrees_node_t 里的键值：

```

public:
    key_type min_key() const //查找最小节点，返回键值，通常是整数
    {
        auto p = ngx_rbtrees_min(get()->root, get()->sentinel);
        return p->key;
    }

```

使用计算偏移量再强制类型转换的方式可以得到完整的树节点对象 T：

```

public:
    T& min() const //查找最小节点，返回实际的节点类型
    {
        auto p = ngx_rbtrees_min(get()->root, get()->sentinel);
        constexpr auto offset = (std::size_t)&((T*)0)->np; //计算偏移量
        return *(T*)((u_char*)(p) - offset); //类型转换
    }

```

更多操作

因为红黑树是标准的二叉树，所以可以很容易为它增加各种算法，例如中序遍历：

```

public:
    template<typename F>
    void traverse(F f) const //中序遍历，传入一个函数对象
    {
        do_traverse(get()->root, f); //传入根节点，开始遍历
    }

private:
    template<typename F>
    void do_traverse(node_type* p, F f) const //中序遍历的递归实现
    {
        if(p == get()->sentinel) //遇到哨兵节点则结束
        { return; }

        do_traverse(p->left, f); //中序遍历左子树
    }

```

```

    f(p);
    do_traverse(p->right, f);
}

```

//使用函数对象访问节点
//中序遍历右子树

类型别名

我们还可以利用 C++11 的类型别名特性定义一些常用的红黑树类型：

```

//标准的红黑树，插入方法使用 ngx_rbtrees_insert_value
template<typename T, ngx_rbtrees_node_t T::* np>
using NgxValueTree = NgxRbtrees<T, np, ngx_rbtrees_insert_value>;

//定时器红黑树，插入方法使用 ngx_rbtrees_insert_timer_value
template<typename T, ngx_rbtrees_node_t T::* np>
using NgxTimerTree = NgxRbtrees<T, np, ngx_rbtrees_insert_timer_value>;

//ngx_event_t 结构体专用红黑树，指定了节点成员函数指针&ngx_event_t::timer
using NgxEventTimerTree = NgxTimerTree<
    ngx_event_t, &ngx_event_t::timer>;

```

用法

首先我们定义一个树节点，注意必须要有一个 ngx_rbtrees_node_t 成员：

```

struct Xinfo
{
    int id;
    ngx_rbtrees_node_t node;
};

```

//自定义的红黑树节点
//节点的自有数据
//必需的红黑树节点成员

示范 NgxRbTree 用法的代码如下：

```

ngx_rbtrees_t      tree;
ngx_rbtrees_node_t sentinel;

typedef NgxValueTree<Xinfo, &Xinfo::node> tree_type;

tree_type::init(tree, sentinel);
tree_type t(tree);
assert(t.empty());

Xinfo arr[5];
for(int i = 0; i < 5; ++i)
{
    arr[i].node.key = i;
}

t.add(arr[4]);
assert(t.min_key() == 4);

```

//红黑树结构体
//红黑树哨兵节点
//简化类型定义
//使用前必须初始化结构体
//C++对象代理红黑树
//此时红黑树为空
//创建 5 个节点对象
//初始化键值
//向红黑树插入一个节点
//查找最小值，为 4

```

assert(t.min().node.key == 4);

t.add(arr[2]);                    //再插入一个节点
assert(t.min_key() == 2);        //查找最小值,此时为2
t.add(arr[3]);
assert(t.min_key() == 2);        //查找最小值,仍然是2

t.add(arr[1]);                    //继续插入节点
t.add(arr[0]);
assert(t.min_key() == 0);        //此时最小值是0

t.del(arr[0]);                    //删除一个节点
assert(t.min_key() == 1);        //此时最小值是1

t.traverse([])(tree_type::node_type * p) //中序遍历红黑树
{    cout << p->key << ", "; }          //输出“1,2,3,4,”

```

4.5 缓冲区

作为 Web 服务器, Nginx 要频繁地收发处理大量的数据, 这些数据有时是连续的内存块, 有时是多个分散内存块, 甚至有时数据过大, 内存无法存放, 只能保存成磁盘文件。

ngx_str_t 结构可以表示内存块, 但它过于简单, 不能应对这些复杂的场景, 所以 Nginx 实现了 ngx_buf_t 和 ngx_chain_t 结构, 专门描述数据缓冲区。

本节介绍 ngx_buf_t, 稍后的 4.6 节介绍 ngx_chain_t。

4.5.1 结构定义

ngx_buf_t 表示一个单块的缓冲区, 既可以是内存也可以是文件。它的结构比较复杂, 可以分成两个部分: 缓冲区信息和标志位信息, 下面的介绍省略了一些不太常用的数据成员(如 shadow、recycled 等)。

缓冲区信息

ngx_buf_t 的缓冲区信息定义如下:

```

// 定义在 core/nginx_buf.h
typedef void *          ngx_buf_tag_t;    //void*类型定义
typedef struct ngx_buf_s ngx_buf_t;      //struct 类型定义

struct ngx_buf_s {

```

```
u_char*      pos;           //内存数据的起始位置
u_char*      last;          //内存数据的结束位置
off_t        file_pos;      //文件数据的起始偏移量
off_t        file_last;     //文件数据的结束偏移量

u_char*      start;         //内存数据的上界
u_char*      end;           //内存数据的下界
ngx_buf_tag_t tag;          //void*指针,可以是任意关联对象
ngx_file_t*  file;          //存储数据的文件对象

...                               //标志位信息
};
```

因为Nginx里的缓冲数据可能在内存或者磁盘文件中,所以 ngx_buf_t 使用 pos/last 和 file_pos/file_last 来指定数据在内存或者文件中的具体位置,究竟数据是在哪里则 要由后面的标志位信息来确定。^①

start 和 end 两个成员变量标记了数据所在内存块的边界,如果内存块是可以修改的, 那么在操作时必须参考这两个成员防止越界。

tag 是一个比较特殊的成员,它的类型是 void*, 用户可以关联任意数据,在代码中任 意解释,通常它指向的是使用该缓冲区的对象。

ngx_buf_t 的内存布局如图 4-5 所示。

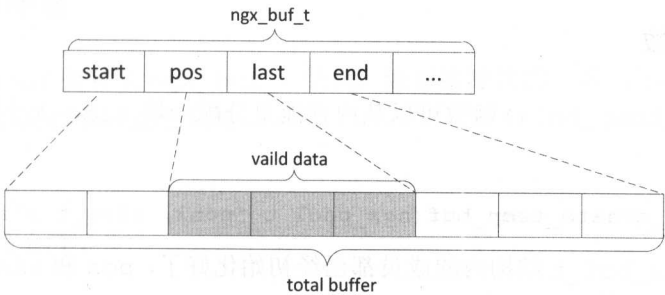


图 4-5 ngx_buf_t 的内存布局

标志位信息

ngx_buf_t 的标志位信息都是 bool 值,使用“位域”的方式以节约内存:

```
struct ngx_buf_s {
```

① 本书暂不介绍 Nginx 里的文件相关操作,读者可参看 core/nginx_file.*。


```

...                                     //缓冲区信息

unsigned        temporary:1;           //内存块临时数据，可以修改
unsigned        memory:1;              //内存块数据，不允许修改
unsigned        mmap:1;                //内存映射数据，不允许修改

unsigned        in_file:1;             //缓冲区在文件里
unsigned        flush:1;               //要求 Nginx 立即输出本缓冲区
unsigned        sync:1;                //要求 Nginx 同步操作本缓冲区
unsigned        last_buf:1;            //最后一块缓冲区
unsigned        last_in_chain:1;       //链里的最后一块缓冲区
unsigned        temp_file:1;           //缓冲区在临时文件里
};

```

这些标志位的含义都比较好理解，但 `last_buf` 和 `last_in_chain` 存在一点小差异：前者是整个处理过程中的最后一块缓冲区，标志着 TCP/HTTP 请求处理的结束；而后者是当前数据块链（`ngx_chain_t`）里的最后一块，之后可能还会有数据需要处理。

从 `ngx_buf_t` 的定义可以看到，一个有数据的缓冲区不是在内存里，就是在文件里，所以内存标志位成员变量（`temporary/memory/mmap`）和文件标志成员变量（`in_file/temp_file`）不能全为 0，否则 Nginx 会认为这是个特殊（`special`）或无效的缓冲区。

如果缓冲区既不在内存也不在文件里，那么它就不含有有效数据，只起到控制作用，例如刷新（`flush`）或者同步（`sync`）。

4.5.2 操作函数

`ngx_create_temp_buf()` 函数可以从内存池里分配一块 `size` 大小的缓冲区，它的声明是：

```
ngx_buf_t *ngx_create_temp_buf(ngx_pool_t *pool, size_t size);
```

函数返回的 `ngx_buf_t` 结构内的成员都已经初始化好了，`pos` 和 `last` 都指向内存块的首位置，表示空缓冲区，而 `temporary` 标志位是 1。

我们也可以直接从内存池创建一个 `ngx_buf_t` 结构，然后手工指定它的成员，关联到已经存在的内存，Nginx 为此提供了两个便捷宏：

```
#define ngx_alloc_buf(pool) ngx_palloc(pool, sizeof(ngx_buf_t))
#define ngx_calloc_buf(pool) ngx_pcalloc(pool, sizeof(ngx_buf_t))
```

两个函数宏可以检查多个标志位，确定缓冲区是否在内存里：

```
#define ngx_buf_in_memory(b) (b->temporary || b->memory || b->mmap)
```

```
#define ngx_buf_in_memory_only(b)    (ngx_buf_in_memory(b) && !b->in_file)
```

起控制作用的特殊缓冲区可以用 `ngx_buf_special()` 判断:

```
#define ngx_buf_special(b)           \
    ((b->flush || b->last_buf || b->sync) \
     && !ngx_buf_in_memory(b) && !b->in_file)
```

函数宏 `ngx_buf_size()` 计算缓冲区的大小, 会根据是否在内存里使用恰当的指针:

```
#define ngx_buf_size(b)              \
    (ngx_buf_in_memory(b) ? (off_t) (b->last - b->pos): \
     (b->file_last - b->file_pos))
```

拷贝内存数据时我们可以直接使用标准 C 函数 `memcpy()`, 但 Nginx 自定义了一个函数宏 `ngx_cpymem`, 接口与 `memcpy()` 相同, 不过它返回的是拷贝数据后的终点位置, 在连续复制多段数据时很方便:

```
// 定义在 core/nginx_string.h
#define ngx_cpymem(dst, src, n)      (((u_char *) memcpy(dst, src, n)) + (n))
```

Nginx 也用函数宏封装了设置内存的 `memset()` 函数:

```
// 定义在 core/nginx_string.h
#define ngx_memzero(buf, n)          (void) memset(buf, 0, n)
#define ngx_memset(buf, c, n)       (void) memset(buf, c, n)
```

4.5.3 C++缓冲区

我们使用 `NgxBuf` 类封装 `ngx_buf_t` 结构, 创建缓冲区的 `ngx_create_temp_buf()` 函数则在 `NgxPool` 里封装, 这里从略。

类定义

`NgxBuf` 的定义如下:

```
class NgxBuf final : public NgxWrapper<ngx_buf_t>
{
public:
    typedef NgxWrapper<ngx_buf_t>    super_type;    //简化类型定义
    typedef NgxBuf                   this_type;
public:
    NgxBuf(const NgxPool& p, std::size_t n):        //从内存池创建缓冲区
        super_type(p.buffer(n))
    {}
```

```

    ngx_buf_t* buf):super_type(buf)           //代理一个缓冲区
    {}

    ~ngx_buf_t() = default;                    //默认析构函数
    ...                                         //成员函数见后
};

```

基本操作

ngx_buf_t 简单封装了 4.5.2 节里操作 ngx_buf_t 的几个函数宏：

```

public:
    bool memory() const                        //缓冲区在内存
    {
        return ngx_buf_in_memory(get());
    }

    bool memoryonly() const                   //缓冲区不在文件
    {
        return ngx_buf_in_memory_only(get());
    }

    bool special() const                      //是特殊的缓冲区
    {
        return ngx_buf_special(get());
    }
public:
    std::size_t size() const                  //缓冲区的大小
    {
        return ngx_buf_size(get());
    }

```

要检查或设置其他的标志位可以直接用重载的“operator->”访问 ngx_buf_t 的成员。

缓冲区操作

如果缓冲区在内存里，可以直接使用 ngx_buf_t 的 pos/last 等成员操作缓冲区，但把操作封装为函数会更好：

```

public:
    void range(u_char* a, u_char* b) const    //使用两个指针设置缓冲区
    {
        get()->pos = a;
        get()->last = b;
        get()->memory = true;                 //设置内存标志
    }

```

```

void boundary(u_char* a, u_char* b) const    //设置缓冲区的边界
{
    get()->start = a;
    get()->end = b;
}

void range(ngx_str_t* s) const              //使用 ngx_str_t 设置缓冲区
{
    range(s->data, s->data + s->len);
    boundary(s->data, s->data + s->len);
}

```

无参的 `range()` 和 `boundary()` 函数以 `ngx_str_t` 形式返回内存块的可用区域和边界，但使用时需要小心，它们并不是真正的字符串，只是一块内存区域：

```

ngx_str_t range() const                    //返回缓冲区内的有效数据
{
    return ngx_str_t{get()->last - get()->pos, get()->pos};
}
ngx_str_t boundary() const                 //返回缓冲区的边界
{
    return ngx_str_t{get()->end - get()->start, get()->start};
}

```

把 `ngx_buf_t` 置为最后一块缓冲区是一个很常用的操作，所以我们封装为成员函数 `last()` 和 `finish()`：

```

bool last() const                          //是否是最后一块缓冲区
{
    return get()->last_buf || get()->last_in_chain;
}

void finish(bool flag = true) const         //置为最后一块缓冲区
{
    get()->last_buf = flag;                  //设置标志位
    get()->last_in_chain = flag;
}

```

NgxBuf 的用法的示范代码可参见 4.6.5 节。

4.6 数据块链

在处理 TCP/HTTP 请求时会经常创建多个缓冲区来存放数据，Nginx 把缓冲区块简单地

组织为一个单向链表，使用 `ngx_chain_t` 结构描述。

4.6.1 结构定义

`ngx_chain_t` 把多个分散的 `ngx_buf_t` 连接为一个顺序的数据块链，定义如下：

```
// 定义在 core/nginx_core.h
typedef struct ngx_chain_s  ngx_chain_t;           //struct 类型定义

// 定义在 core/nginx_buf.h
struct ngx_chain_s {
    ngx_buf_t*    buf;                //缓冲区指针
    ngx_chain_t*  next;              //下一个链表节点
};
```

`ngx_chain_t` 的结构很简单，就是经典的链表节点。如果这个节点是链表的尾节点，就必须要把 `next` 置为 `nullptr`，表示链表结束，否则会导致未定义错误。^①

4.6.2 操作函数

与 `ngx_queue_t` 类似，虽然 `ngx_chain_t` 的结构简单，但它表示了复杂的链表结构，可以对它执行很多操作，本节只介绍最基本的几个。

`ngx_alloc_chain_link()` 和 `ngx_free_chain()` 用来从内存池里获取释放 `ngx_chain_t` 对象，声明是：

```
ngx_chain_t *ngx_alloc_chain_link(ngx_pool_t *pool);
#define ngx_free_chain(pool, cl)
```

由于 `ngx_chain_t` 在 Nginx 里应用得很频繁，所以 Nginx 对此进行了优化。在内存池里保存了一个空闲 `ngx_chain_t` 链表，分配时从这个链表里摘取，释放时再挂上去。^②

函数 `ngx_create_chain_of_bufs()` 可以一次创建多个缓冲区，返回一个连接好的数据块链表：

```
typedef struct {
    ngx_int_t    num;                //缓冲区的数量，即节点数量
    size_t       size;              //缓冲区的大小
```

① 这是一个很容易犯的错误，即使是 Nginx 自己有时候也会失误（出现在 1.11.11 里的一个 bug）。

② 注意，使用 `ngx_alloc_chain_link()` 内部调用的是 `ngx_palloc()`，获得的 `ngx_chain_t` 对象的 `buf` 和 `next` 指针可能是任意值，不能假设 `next` 指针是空指针。

```

} ngx_bufs_t;                                     //创建链表的参数结构

ngx_chain_t *ngx_create_chain_of_bufs(ngx_pool_t *pool, ngx_bufs_t *bufs);

ngx_bufs_t 是函数 ngx_create_chain_of_bufs() 的参数, 确定了创建的缓冲区数
量和大小。

```

4.6.3 C++节点

我们仍然把 `ngx_chain_t` 分解为节点、迭代器和容器三个概念, 不同的 C++ 类封装不同种类的操作。

类定义

`NgxChainNode` 封装了 `ngx_chain_t` 的节点操作, 由于 `ngx_chain_t` 结构简单, 所以比 `NgxQueueNode` 要少很多代码:

```

class NgxChainNode final : public NgxWrapper<ngx_chain_t>
{
public:
    typedef NgxWrapper<ngx_chain_t> super_type;      //简化类型定义
    typedef NgxChainNode          this_type;

public:
    NgxChainNode(ngx_chain_t* c):super_type(c)
    {}

    ~NgxChainNode() = default;

public:
    ...                                              //成员函数见后
};

```

操作函数

`NgxChainNode` 把对 `next` 指针的操作分解为三个函数, 明确地表示检查尾节点、链接节点和结束链表 (置尾节点):

```

public:
    bool last() const                                //是否是最后一个节点
    {
        return !get()->next;                        //检查空指针
    }

    void link(ngx_chain_t* c) const                 //链接一个节点
    {

```

```

        get()->next = c;                                //next 指针赋值
    }

    void finish() const                                //结束链表
    {
        link(nullptr);                                  //链接到空指针，即结束
    }

```

成员函数 `data()` 和 `set()` 操作 `ngx_chain_t` 的 `buf` 成员, `data()` 返回 `NgxBuf` 对象, `set()` 设置 `buf` 指针:

```

public:
    NgxBuf data() const                                //获取节点的缓冲区
    {
        get()->buf;                                     //隐式转换为 NgxBuf 对象
    }

    void set(ngx_buf_t* ptr) const                      //设置缓冲区指针
    {
        get()->buf = ptr;
    }

```

内存池创建节点

在 `NgxPool` 里可以封装 `ngx_chain_t` 的创建工作:

```

class NgxPool final : public NgxWrapper<ngx_pool_t>
{
public:
    ...                                                //其他成员函数
    ngx_chain_t* chain() const                        //创建一个空节点
    {
        auto p = ngx_alloc_chain_link(get());        //内存池获取节点

        NgxException::require(p);                    //检查空指针
        p->next = nullptr;                            //设置 next 为空指针

        return p;                                     //返回节点指针
    }

    ngx_chain_t* chain(const ngx_bufs_t& bufs) const  //创建多个节点
    {
        auto p = ngx_create_chain_of_bufs(get(),
            const_cast<ngx_bufs_t*>(&bufs));

        NgxException::require(p);                    //检查空指针
    }

```

```

        return p;                                //返回节点指针
    }
};

```

chain() 函数以重载的方式同时支持了创建单个和多个链表节点的情形, 为了避免 next 指针不确定的隐患, 它总把 next 置为 nullptr。

4.6.4 C++迭代器

ngx_chain_t 的迭代器实现与 ngx_list_t 的迭代器 NgxListIterator 类似, 因为它们都是单向链表, 但 ngx_chain_t 的遍历操作显然要容易得多, 只需指针后移即可。

类定义

NgxChainIterator 使用 single_pass_traversal_tag 实现单向迭代器:

```

class NgxChainIterator final:
    public boost::iterator_facade<
        NgxChainIterator, NgxChainNode,                //返回代理对象
        boost::single_pass_traversal_tag>
{
public:
    typedef boost::iterator_facade<...>      super_type;    //简化类型定义
    typedef typename super_type::reference reference;
public:
    NgxChainIterator(ngx_chain_t* c) : m_p(c)              //获取节点指针
    {}

    NgxChainIterator() = default;
    ~NgxChainIterator() = default;
private:
    ngx_chain_t* m_p = nullptr;                      //节点指针
    mutable NgxChainNode m_proxy{m_p};                //代理对象
public:
    ...                                                //成员函数见后
};

```

迭代器核心操作

NgxChainIterator 需要实现单向迭代器必需的 dereference()、increment() 和 equal() 三个操作:

```

private:
    friend class boost::iterator_core_access;          //必需的友元声明

```



```

reference dereference() const //解引用操作, 返回代理对象
{
    m_proxy = m_p; //拷贝赋值节点
    return m_proxy; //返回节点对象的代理引用
}

void increment() //前进迭代器
{
    if(!m_p) //检查空指针
    { return; }

    m_p = m_p->next; //指向下一个节点
}

bool equal(NgxChainIterator const& o) const //比较迭代器
{
    return m_p == o.m_p;
}

```

注意: `NgxChainIterator` 的解引用操作返回的是一个代理对象 `NgxChainNode`, 而不是 `ngx_chain_t*`, 所以更容易使用。

`NgxChainIterator` 还实现了其他一些操作, 请参考 GitHub 资源。

4.6.5 C++数据块链

`NgxChain` 封装了 `ngx_chain_t` 链表:

```

class NgxChain final : public NgxWrapper<ngx_chain_t>
{
public:
    typedef NgxWrapper<ngx_chain_t> super_type; //简化类型定义
    typedef NgxChain this_type;
public:
    NgxChain(ngx_chain_t* c):super_type(c) //代理链表节点
    {}

    ~NgxChain() = default;
public:
    typedef NgxChainIterator iterator; //迭代器类型定义
    typedef const iterator const_iterator;

    iterator begin() const //产生迭代器起点
    {

```

```

        return iterator(get());
    }

    iterator end() const //产生迭代器终点
    {
        return iterator();
    }
public:
    std::size_t size() const //遍历链表，计算总长度
    {
        std::size_t len = 0;

        for(auto& c : *this)
        {
            len += c.data().size(); //获取节点的 buf 长度
        }

        return len;
    }

```

链表较常用的操作是向尾部添加节点，所以 `NgxChain` 实现了两个函数：

```

public:
    ngx_chain_t* tail() const //查找末尾节点
    {
        auto p = get(); //头节点

        for(;p->next;p = p->next); //查找空指针

        return p; //返回 next 为空的尾节点
    }

    void append(ngx_chain_t* ch) const //向末尾追加节点
    {
        tail()->next = ch; //调整 next 指针
    }

```

下面的代码综合示范了 `NgxBuf` 和 `NgxChain` 的用法：

```

NgxPool pool(r); //内存池对象
NgxBuf buf(pool.buffer(10)); //创建一个10字节缓冲区对象

assert(buf.memory() && buf.memoryonly()); //在内存里，不在文件里
assert(buf.range().len == 0); //无有效数据
assert(buf.boundary().len == 10); //缓冲区大小是10字节

```

```

ngx_memset(buf.range().data, 'a', 5);           //写入 5 个字节数据
buf->last += 5;                                  //移动缓冲区数据指针
assert(buf.range().len == 5);                   //有 5 个字节的有效长度
cout << "str=" << buf.range() << endl;         //输出缓冲区数据

NgxChain ch = pool.chain();                     //创建一个缓冲区链

auto node = *ch.begin();                        //获取第一个节点
assert(node.last());                           //next 指针已经置为空
node.set(buf);                                  //设置缓冲区数据
cout << node.data().range() << endl;           //输出缓冲区数据

NgxChain ch2 = pool.chain(ngx_bufs_t{2, 10});  //创建另一个链表，两个节点
node.link(ch2);                                //链接到第一个链表

for(auto&x : ch)                                //遍历链表，可以用 for
{
    auto b = x.data();                          //获得缓冲区代理
    ngx_memset(b->pos, 'b', 5);                 //设置缓冲区数据
    b->last = b->pos + 5;                       //移动缓冲区数据指针
    cout << b.range() << ";";                 //输出缓冲区数据
}

```

GitHub 上的 `NgxChain` 还有更多的接口，如 `trace()`、`clear()`、`copy()` 等，读者可自行参考。

4.7 键值对

键值对 (key-value pair) 是一种映射关系，可以把一个值映射到另一个值。C++ 使用 `std::pair` 来表示，并且使用 `std::map` 和 `std::unordered_map` 来存储这样的数据，而 Nginx 则提供了两个结构：`ngx_keyval_t` 和 `ngx_table_elt_t`，再结合 `ngx_array_t` 或 `ngx_list_t` 应用在不同的场景。

这两个结构比较简单，所以本书不使用 C++ 封装。

4.7.1 简单键值对

`ngx_keyval_t` 是一个简单的键值对结构，主要用在 Nginx 的配置解析环节，保存配置文件里成对的配置，定义如下：

```

// 定义在 core/nginx_string.h
typedef struct {

```

```

    ngx_str_t    key;                                //键
    ngx_str_t    value;                              //值
} ngx_keyval_t;

```

在 Nginx 里，通常使用 `ngx_array_t` 来存储 `ngx_keyval_t`，相当于：

```
typedef NgxArray<ngx_keyval_t>    NgxKvArray;
```

4.7.2 散列表键值对

`ngx_table_elt_t` 与 `ngx_keyval_t` 很类似，但多了两个成员：

```

// 定义在 core/nginx_hash.h
typedef struct {
    ngx_uint_t    hash;                                //散列（哈希）标记
    ngx_str_t     key;                                //键
    ngx_str_t     value;                              //值
    u_char        *lowcase_key;                      //key 的小写字符串指针
} ngx_table_elt_t;

```

`ngx_table_elt_t` 主要用来表示 HTTP 头部信息，例如 “Server: nginx” 这样的字符串对应到 `ngx_table_elt_t`，就是 `key="Server"`，`value="nginx"`。

成员 `hash` 是一个散列标记，Nginx 使用它在散列表结构里快速查找数据。可以简单地把它置为非零值（通常是 1），也可以使用下面的两个函数计算散列值：

```

// 定义在 core/nginx_hash.h
ngx_uint_t ngx_hash_key(u_char *data, size_t len);    //计算散列值
ngx_uint_t ngx_hash_key_lc(u_char *data, size_t len); //小写后再计算

```

成员 `lowcase_key` 指向了一个全小写的字符串，在大小写无关比较时可避免重复计算。

函数 `ngx_hash_strlow()` 可以在小写化的同时计算出散列值：

```
ngx_uint_t ngx_hash_strlow(u_char *dst, u_char *src, size_t n);
```

Nginx 在处理 HTTP 请求时使用 `ngx_list_t` 存储了 HTTP 头部信息，相当于：

```
typedef NgxList<ngx_table_elt_t>    NgxHeadersList;
```

4.8 总结

本章详细介绍了 `ngx_array_t`、`ngx_list_t` 等 Nginx 里的高级数据结构，这些数据结构类似 C++ 里的泛型标准容器，可以存储各种类型的元素。它们管理组织 Web 服务器里繁多的数据，是 Nginx 宫殿里的立柱、横梁等重要建筑构件。

`ngx_array_t` 是一个泛型动态数组，它结构简单，存取效率高，与 C++ 里的 `vector` 一样，是最容易使用的一个数据容器。

`ngx_list_t` 是一个有优化的单向链表。传统的链表一个节点只存储一个元素，因而存储成本高，而 `ngx_list_t` 的节点里是一个数组，可以存放多个元素，很好地摊平了链表指针的额外成本，节约了内存。

`ngx_queue_t` 定义了双向链表的前后指针，它必须以数据结构的成员方式使用，是一种“侵入式”容器。但侵入式容器的用法没有非侵入式容器那么简单，Nginx 使用 `offsetof` 宏以一种巧妙的方式可以从成员变量的地址访问到整个元素。

`ngx_rbtrees_t` 是经典的红黑树结构，它是一种自平衡二叉树，查找、插入、删除等操作都很高效，Nginx 使用它实现了定时器机制来处理超时事件。

`ngx_buf_t` 和 `ngx_chain_t` 描述了 Nginx 里的缓冲区，可以表示连续或分散的缓冲数据。这些数据既可以在内存中，也可以在磁盘文件里，在处理 TCP/HTTP 请求时需要经常使用这两个数据结构接收和发送数据。

最后我们简单讨论了 Nginx 里的键值对结构，它们通常结合前面的几个容器来表示配置信息或者 HTTP 头部信息。

Nginx 提供的数据结构虽然定义比较简单，但接口操作多，而且没有很好的分类，很容易误用或滥用。本章使用 C++ 的封装特性分解了这些数据结构的操作，提取出迭代器、节点和容器等概念，把数据结构和操作绑定在一起，给出了更安全可靠的接口，用法更简单，也不容易出错。

Nginx 里还有其他一些高级数据结构，如散列表、基数树等，本章并没有介绍，主要是因为它们通常在 Nginx 内部使用，编写实际的 TCP/HTTP 应用模块时较少遇到，而且 C++ 中可以在标准库或 Boost 库里找到等价且更好的容器，感兴趣的读者可以阅读 Nginx 源码自行研究。

第 5 章

Nginx开发综述

在前面的章节中我们学习了 Nginx 里的各种数据结构，对 Nginx 源码有了基本的认识，下面就来使用 C/C++编写真正的 Nginx 模块。

由于 Nginx 模块开发涉及的要素很多，很难一下子讲解清楚，所以本章从一个简单的例子开始，通过实际的代码来让读者熟悉模块开发的基本步骤。有了这些初步认识后再详细介绍编译脚本、配置文件解析、模块的定义和操作函数等较深入的细节。

5.1 最简单的模块

本节我们将利用前两章的 C++封装类如 NgxPool、NgxLog 等实现一个简单的 http 模块，它基本上什么也不做，只是在控制台输出一个字符串并记录运行日志。^①

在编写模块前，我们先对这个模块提出三个问题：

- 是否需要在配置文件里配置？配置指令是什么？有什么样的参数？
- 怎样使用 Nginx 框架？怎样访问配置参数？怎样处理 TCP/HTTP 请求？
- 如何编译集成进 Nginx？

接下来的小节将逐步解决这三个问题，完成模块的开发。

① 它并不是“最”简单的 Nginx 模块，真正最简单的模块应该是什么也不做，没有任何配置也不介入处理流程，但这样的模块对我们来说毫无意义。

5.1.1 模块设计

不管 Nginx 模块是简单还是复杂，是 http 还是 stream、event、core 模块，我们都需要对它进行设计，回答刚才提出的三个问题。设计可以只存在于我们的头脑中，或者是形成正式的书面文档。

对于这个简单功能的模块，我们的设计如下：

- 模块的名字是 `ndg_test_module`。
- 配置指令是 `ndg_test on|off`，开关模块的功能只能在 `location` 里配置。
- 使用 `ngx_command_t` 和相关函数解析配置指令。
- 使用 `ngx_http_module_t` 定义功能函数，创建配置数据并初始化。
- 使用 `ngx_module_t` 定义模块。
- 不直接处理 HTTP 请求，只在 URL 重写阶段里执行。
- 根据配置指令的 `on|off` 决定输出字符串的内容。
- 编写 `config` 脚本，用 “`--add-module`” 静态链接选项集成进 Nginx。
- 为了简单起见，所有代码都实现在一个 `cpp` 里，名字是 `ModNdgTest.cpp`。

这只是个很粗略的设计，但目前来说基本够用，下面就开始具体的实现。

5.1.2 配置解析

`ndg_test_module` 在 Nginx 配置文件里的形式是：

```
location /test {                                #一个 location
    ndg_test on;                                #启用模块的功能
    ...                                          #其他配置指令
}
```

解析这个配置指令要定义配置数据结构、指令数组和管理函数。

配置的数据结构

我们必须定义一个含有对应信息的结构体，用来存储配置数据。这里结构体命名为 `NdgTestConf`：^①

```
struct NdgTestConf final                        //final 关键字，禁止被继承
{
```

^① Nginx 对配置结构体的命名有自己的规范，通常的形式是 `ngx_http_module_xxx_conf_t`，但我们使用的是 C++，所以不必受它的约束。

```
    ngx_flag_t enabled = ngx_nil;           //标志变量，构造时初始化
};
```

NdgTestConf 里定义了 enabled 变量，它的类型是 ngx_flag_t，可以把配置文件里的 on|off 信息转换为 1|0 保存。

NdgTestConf 另外一项重要的工作是初始化，使用了 C++11 的新特性，明确设置为 UNSET，这样在后续的流程中 Nginx 才能正确处理。

配置的解析

有了配置的数据结构仅仅是个开始，它还没有与配置指令关联起来。

Nginx 提供结构 ngx_command_t 来实现配置指令解析，每条指令对应一个 ngx_command_t 对象。所有的 ngx_command_t 对象放在一个数组里，最后需要用宏 ngx_null_command 表示数组定义结束。

因为 ndg_test_module 只有一条指令，所以指令数组比较简单，代码如下：

```
static ngx_command_t ndg_test_cmds[] =      //配置指令数组
{
    {
        ngx_string("ndg_test"),             //指令的名字
        NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,    //指令的作用域和类型
        ngx_conf_set_flag_slot,             //解析函数指针
        NGX_HTTP_LOC_CONF_OFFSET,           //数据的存储位置
        offsetof(NdgTestConf, enabled),      //数据的具体存储变量
        nullptr                              //暂无须关心
    },

    ngx_null_command                         //空对象，结束数组
};
```

ngx_command_t 共有六个成员，其具体含义将在第 6 章讲解，这里只做简要的说明：

- 指令的名字 ndg_test，它就是出现在配置文件里的指令；
- 指令只能在 location 里出现，可接受的参数是 on|off；
- 使用 Nginx 提供的标准函数 ngx_conf_set_flag_slot 解析指令；
- 整个数据结构存储在 http/location 作用域；
- 使用宏 offsetof 获取变量的地址，供解析函数使用；
- 数组的最后必须是 ngx_null_command，起到哨兵的作用，相当于 '\0'。Nginx 使用它来标识命令数组的结束。

创建配置数据

Nginx 要求模块自己分配配置数据结构的内存，所以我们还需要编写一个 `create()` 函数，在配置解析时使用内存池创建对象：

```
static void* create(ngx_conf_t* cf)           //创建配置数据结构
{
    return NgxPool(cf).alloc<NdgTestConf>(); //由内存池分配内存，构造对象
}
```

`create()` 的函数参数 `ngx_conf_t` 代表了 Nginx 在解析配置指令时的环境 (Context)，它有一个内存池成员 `pool`，所以我们可以构造 `NgxPool` 对象然后分配内存。

如果对象里的成员没有在构造时初始化，那么必须在这里完成初始化，设置为 `UNSET`。

`create()` 函数将在 5.1.4 节里通过 `ngx_http_module_t` 插入到 Nginx 框架，被 Nginx 调用。

5.1.3 处理函数

基本的 TCP/HTTP 请求处理流程我们都很熟悉：接收数据（例如请求头和请求体），检查权限，然后产生响应内容返回给客户端，最后记录日志。

Nginx 在框架的级别上仔细梳理了 TCP/HTTP 请求处理流程，提炼出了权限检查、内容产生、记录日志等多个明晰的阶段 (phases, HTTP 处理有 11 个，详细的讲解可参见第 7 章)，简化了开发工作。我们可以在这些阶段里插入自己的功能代码，实现所需的业务逻辑。

`ndg_test_module` 的功能很简单，它只是用来测试，不响应用户请求，所以可以选择任何阶段。依据 5.1.1 的设计，它将在 `rewrite` 阶段执行。

我们需要实现两个函数：实际的处理功能和把它插入到 Nginx 的处理流程框架。^①

处理函数

函数 `handler()` 先读取配置参数，然后根据参数向控制台输出字符串：

```
static ngx_int_t handler(ngx_http_request_t *r)
{
```

① 因为我们使用了 C++ 异常，正确的做法是总使用 `try-catch` 块来捕获异常，防止错误发生——特别是在 Nginx 的请求处理阶段。但那样会导致代码过于冗余，所以本书之后的代码基本都不写 `try-catch` 块，请读者注意。

```

auto cf = reinterpret_cast<NdgTestConf*>(           //获取配置数据
    ngx_http_get_module_loc_conf(r, ndg_test_module));

NgxLogError(r).print("hello c++");                //记录运行日志

if (cf->enabled)                                    //检查配置参数
{
    std::cout << "hello nginx" << std::endl;      //输出字符串
}
else
{
    std::cout << "hello disabled" << std::endl;    //输出字符串
}

return NGX_DECLINED;                               //执行成功但未处理
}

```

因为我们在配置解析时指定了参数 `NGX_HTTP_LOC_CONF_OFFSET`，所以要使用 Nginx 提供的函数宏 `ngx_http_get_module_loc_conf` 来获取模块存储在 location 域里的配置信息。它返回的是 `void*` 指针，需要转型为模块自己的配置数据类型。

之后的处理逻辑很简单，首先使用封装类 `NgxLogError` 记录一条日志，然后根据 `enabled` 的 0/1 值输出不同的字符串。

函数的结尾我们必须返回 `NGX_DECLINED` 而不能是 `NGX_OK`，这是因为我们并没有真正地处理数据，Nginx 在收到 `NGX_DECLINED` 返回值后会继续调用后续的模块处理。

注册处理函数

函数 `init()` 将在 Nginx 配置解析阶段调用，向 Nginx 框架注册我们自己的处理函数：

```

static ngx_int_t init(ngx_conf_t* cf)
{
    auto cmcf = reinterpret_cast<ngx_http_core_main_conf_t*>(
        ngx_http_conf_get_module_main_conf(           //注意获取函数
            cf, ngx_http_core_module));                //注意使用的模块

    NgxArray<ngx_http_handler_pt> arr(                  //Nginx 数组
        cmcf->phases[NGX_HTTP_REWRITE_PHASE].handlers); //该阶段的 handler

    arr.push(handler);                                  //加入自己的 handler

    return NGX_OK;                                       //执行成功
}

```

在 Nginx 的模块架构里有一个特别的模块 `ngx_http_core_module`，它负责管理所有

的 http 功能模块。因为 `init()` 发生在配置解析阶段，所以需要另外用一个函数宏 `ngx_http_conf_get_module_main_conf` 获取它的配置结构，然后把处理函数 `handler` 加入到 HTTP 请求处理的 `rewrite` 阶段的 `handler` 数组里。Nginx 在处理 HTTP 请求时会依次调用每个阶段里的所有 `handler`，从而执行我们自己的功能。

5.1.4 模块集成

Nginx 提供两个数据结构 `ngx_http_module_t` 和 `ngx_module_t`，用来集成配置指令解析和处理函数。

集成配置函数

`ngx_http_module_t` 是 http 模块专用的数据结构，包含 8 个在配置解析阶段可被 Nginx 调用的函数指针（见代码里的注释）。对于模块 `ndg_test_module`，我们只实现了其中的两个：

```
static ngx_http_module_t ndg_test_ctx =
{
    nullptr,                //preconfiguration, 解析配置文件前被调用
    init,                   //postconfiguration, 解析配置文件后被调用
    nullptr,                //create_main_conf, 创建 http main 域的配置结构
    nullptr,                //init_main_conf, 初始化 http main 域的配置结构
    nullptr,                //create_srv_conf, 创建 server 域的配置结构
    nullptr,                //merge_srv_conf, 合并 server 域的配置结构
    create,                 //create_loc_conf, 创建 location 域的配置结构
    nullptr,                //merge_loc_conf, 合并 location 域的配置结构
};
```

从代码的注释我们可以知道：函数 `create()` 将在解析配置文件 `location` 时被调用，创建出 `NdgTestConf` 对象，然后存储在 Nginx 的 `location` 内存域里。函数 `init()` 将在所有的配置指令解析完毕后被调用，修改 `ngx_http_core_module` 的 `handler` 数组，注册处理函数。

因为我们的配置指令只出现在 `location` 域里，并不需要合并参数，所以有关 `main` 和 `server` 的函数指针都置为 `nullptr`。

集成配置指令

`ngx_module_t` 是 Nginx 真正定义模块的数据结构，它集成 `ngx_http_module_t` 和 `ngx_command_t` 数组：

```
ngx_module_t ndg_test_module =                //模块定义，注意不是 static
{
```

```

    NGX_MODULE_V1,                //标准的填充宏
    &ndg_test_ctx,                //配置功能函数
    ndg_test_cmds,                //配置指令数组
    NGX_HTTP_MODULE,              //http 模块必需的 tag
    nullptr,                       //init master
    nullptr,                       //init module
    nullptr,                       //init process
    nullptr,                       //init thread
    nullptr,                       //exit thread
    nullptr,                       //exit process
    nullptr,                       //exit master
    NGX_MODULE_V1_PADDING         //标准的填充宏
};

```

ngx_module_t 结构比较复杂，定义了很多 Nginx 框架在加载模块时所必需的信息。其中最重要的就是配置功能函数和配置指令数组，其他的都可以使用预定义宏或者空指针填充。对于 http 模块，需要在配置指令数组后使用宏 NGX_HTTP_MODULE 作为模块的标识。

需要注意的是，ngx_module_t 变量不能是 static 的，这是因为它将被 Nginx 框架所使用，如果声明为 static 那么 Nginx 将无法访问模块对象。

到这里，ndg_test_module 所有的 C++代码就编写完成了。

5.1.5 编译脚本和命令

根据 Nginx configure 的要求，我们要编写一个 Shell 脚本，然后才能用 “--add-module” 或 “--add-dynamic-module” 参数配置编译选项。^①

编译脚本

Nginx 要求模块的 Shell 脚本名字必须是 config，里面使用特定的 Shell 变量告诉 configure 模块相关的信息，再调用 auto/module:

```

#脚本名必须是 config

#模块的名字，只是在 configure 时显示信息用，不必与代码里的相同
ngx_addon_name=ndg_test_module

#模块的类型，http 模块通常使用 HTTP 或 HTTP_FILTER
ngx_module_type=HTTP

#模块在 C/C++代码里的名字，编译链接时使用，允许有多个不同的模块

```

^① Nginx 在 1.9.11 版之前只能在编译时以静态链接的方式加载第三方模块。

```
ngx_module_name=ndg_test_module

#模块的源码，也可以有多个，用空格分隔
ngx_module_srcs="$ngx_addon_dir/ModNdgTest.cpp"

#模块的包含路径，是我们存放 C/C++头文件的位置
ngx_module_incs="$ngx_addon_dir/../../ngxpp"

#重要步骤！调用 Nginx 提供的脚本处理上述 Shell 变量
. auto/module
```

在 config 脚本里必须定义的是 ngx_addon_name、ngx_module_type、ngx_module_name 和 ngx_module_srcs，它们给出了模块最基本的信息，configure 将利用这四个变量产生用于编译的 Makefile 和源码。

因为 ndg_test_module 的代码使用了 C++包装类，所以还需要用 ngx_module_incs 指定编译时的头文件包含路径。

最后我们必须用 “. auto/module” 的形式调用 Nginx 的脚本，才能把模块正确地添加进 Nginx。

编译命令

编写完 config 脚本后，模块的所有开发工作就完成了。只要在 configure 时使用 “--add-module” 或 “--add-dynamic-module” 参数就可以把模块集成进 Nginx 框架。

假设模块的代码和脚本存放在 “~/ngx_cpp_dev/http/test” 下，那么静态链接的配置的命令就是：

```
./configure --add-module=$HOME/ngx_cpp_dev/http/test
```

配置完成后再执行 make && make install，我们的开发就大功告成。

5.1.6 测试验证

为了验证 ndg_test_module 的功能需要修改配置文件，新建一个 location，并使用 ndg_test 指令打开或者关闭模块。

因为模块使用了 cout 输出信息字符串，所以我们需要使用 “daemon off;” 让 Nginx 在前台运行。相关的配置文件片段是：

```
master_process off;                                #禁用进程池，使用单进程
daemon off;                                         #禁止 daemon，前台运行
```

```
...                                     #其他配置

location /test {                       #测试使用的 location 配置
    ndg_test on;                       #打开模块

    alias /usr/local/nginx/html;       #设置静态文件的路径
    index index.html index.htm;       #设置 index 文件
}
```

使用 curl 向 test location 发送测试请求：

```
curl -vo /dev/null 'http://localhost/test/index.html'
```

可以看到 Nginx 在控制台输出字符串 “hello nginx”，查看 error.log 也可以看到打印的运行日志：

```
2017/xx/xx 09:20:12 [error] 31213#0: *1 hello c++, client: 127.0.0.1, ...
```

5.2 开发基本流程

在 5.1 节，我们通过开发一个简单的模块初步熟悉了 Nginx 的模块开发流程。以小见大，任何 Nginx 模块的开发过程都与之类似，都要经过设计、开发、编译、测试验证和调优五个步骤，本节将对此做简略阐述，然后在之后详细讲解 Nginx 的开发知识。

5.2.1 设计

“凡事预则立，不预则废”，这句古语不仅对于 Nginx 模块开发，对于任何软件产品的开发都非常有警示作用。

在模块的设计阶段，我们必须做出模块各方面的总体决策，包括但不限于：

- 模块的数量和名字；
- 模块的配置指令和参数；
- 模块的基本业务逻辑；
- 模块的结构和组成；
- 模块的测试用例。

设计的成果最好以文档的形式保存下来，除基本的备忘作用之外，更重要的是把思维固化下来利于以后的交流、维护和经验总结。

5.2.2 开发

开发是整个 Nginx 模块开发流程中最重要的步骤，它直接实现设计，可以细分成配置指令开发、业务逻辑开发、框架注册和模块集成四个相对较为独立的部分。

配置指令开发

配置文件是 Nginx 的核心，所以这常常是模块开发时最先开始的工作。

我们要依据设计文档定义存储配置信息的数据结构，还有它的创建、解析、合并等操作函数——注意在创建配置数据结构时必须初始化成员为 UNSET，最后用一个 `ngx_command_t` 数组描述这些指令。

如果指令比较简单，那么可以使用 Nginx 框架提供的一些预定义函数简化开发工作。

业务逻辑开发

这是实现模块自身功能的地方，具体的逻辑因模块而异。对于 http 模块来说，通常的步骤是获取配置参数，获取 HTTP 请求头和请求体数据，做一些处理或者转发到外部，得到内容后再设置 HTTP 响应头和响应体，返回给 Nginx 框架合适的错误码。

在处理过程中应该多使用 Nginx 的日志机制，分级别记录运行日志便于排错。

框架注册

Nginx 框架需要知道我们的功能如何介入 TCP/HTTP 请求处理流程，注册动作发生在启动时的解析配置文件阶段。

以 http 模块为例，Nginx 提供了两种注册方式，一种是 5.1 节使用的方式，在完成配置文件解析后修改 `ngx_http_core_module` 的 `phase[xxx].handlers` 数组，插入函数指针；另一种则是在解析到特定指令时直接设置该 `location` 的处理函数，忽略 `phase handlers` 函数数组。

模块集成

有了配置函数、业务逻辑函数和框架注册函数，我们要设置 `ngx_http_module_t` 和 `ngx_module_t` 里的成员变量和成员函数指针，这样 Nginx 框架就可以在配置解析阶段和请求处理阶段正确调用我们的函数，实现模块的功能。

5.2.3 编译

相对于开发来说 Nginx 的模块编译比较简单（当然也可以有复杂的逻辑），只要编写一个 Shell 脚本，并设置 `ngx_module_type`、`ngx_module_name` 等变量的值。

在执行“`configure --add-module/--add-dynamic-module`”时，Nginx 会调用 `config` 脚本，获取里面的变量，然后生成最后用于编译的 Makefile 和 C 源码文件。

5.2.4 测试验证

完成开发和编译步骤后必须要进行测试验证，可以是简单的自我测试，也可以是更正式的功能和性能测试，必须认真对待。测试的基本内容是验证 Nginx 在加入模块后能否正确启动运行、模块能否正确解析配置指令、能否正确完成设想的业务逻辑。

简单的测试工作可以用 `curl` 向 Nginx 发送构造好的 HTTP 请求，检查返回的响应和运行日志是否符合预期。更正式一些就需要使用脚本语言编写测试用例，实现自动化测试。

测试过程中一旦发生了错误，就要返回开发和编译步骤，查找错误原因，排除故障后再重新进行测试，直至所有测试用例通过。

5.2.5 调优

如果模块的功能比较简单，代码量较少，通常开发流程进入到第 4 步测试验证就可以结束了，不需要调优。但如果模块的业务比较复杂，网络吞吐量很大，调优就非常有必要，可以让模块运行得更加快速稳定，资源利用率更高。

我们可以使用一些工具进行压力测试，或者观察线上真实环境，使用 `top`、`iostat`、`sar`、`systemtap` 等工具检查 Nginx 的内存、CPU、磁盘以及网络使用状况，结合日志分析，找出程序运行的瓶颈，调整 Nginx 相关的配置参数，或者优化模块的代码，然后再观察 Nginx 参数是否有改善，反复迭代，直至达到预期的目标。

调优是一个没有固定模式的工作，需要开发和运维共同努力，还有实际经验的积累。

5.2.6 流程图

图 5-1 简单描述了 Nginx 开发的基本流程。

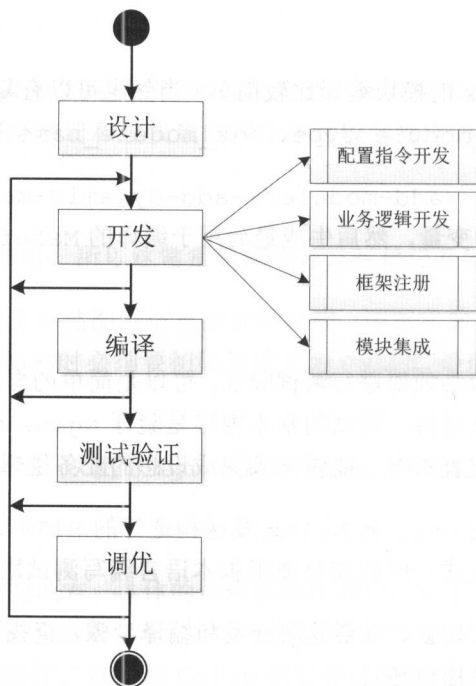


图 5-1 Nginx 开发的基本流程

5.3 编译脚本

Nginx 并没有使用 autoconf、cmake 等流行的编译配置工具，而是使用纯 Shell 脚本打造了自己的配置工具，优点是灵活且兼容性高，不会受到工具和操作系统的限制。

我们只需具备基本的 Shell 编程知识就可以很容易地理解编译机制，实现编译脚本。

5.3.1 运行机制

当使用参数 “--add_module” 或 “--add-dynamic-module” 时，configure 脚本会把所有的第三方模块加入到内部的脚本变量里，代码是（位于 auto/options）：

```

--add-module=*)          NGX_ADDONS="$NGX_ADDONS $value" ;;
--add-dynamic-module=*)  DYNAMIC_ADDONS="$DYNAMIC_ADDONS $value" ;;

```

之后，configure 调用 auto/modules，执行每个模块所在目录里的 config 脚本，部分代码摘录如下：

```

if test -n "$NGX_ADDONS"; then                                #检查是否有第三方模块，静态链接

```

```

for ngx_addon_dir in $NGX_ADDONS          #循环加入第三方模块
do
    echo "adding module in $ngx_addon_dir"  #输出模块的目录信息

    ngx_module_link=ADDON                  #设置模块链接方式，静态链接
    if test -f $ngx_addon_dir/config; then  #是否有 config 脚本
        . $ngx_addon_dir/config            #在这里执行 config 脚本
        echo " + $ngx_addon_name was configured"
    fi
done
fi

```

config 脚本里主要做的工作就是设置 Shell 变量，当然，也可以不仅限于此，加入复杂的判断逻辑，检查编译器版本、外部依赖库，但最后仍要设置变量，这是 Nginx 的编译机制要求的。

5.3.2 使用的变量

在 config 里可以操作很多预定义的变量，最基本的是以下四个：

- ngx_addon_name : 模块的名字，仅供显示用，与编译无关；
- ngx_module_type : 模块的类型，常用的取值是 STREAM、HTTP、HTTP_FILTER、HTTP_AUX_FILTER 等；
- ngx_module_name : 模块在 C/C++ 代码里的名字，用于编译链接；
- ngx_module_srcs : 模块的源码文件，可以有多个，用空格分隔。

还有另外一些变量，用来设置编译的其他参数：

- ngx_module_incs : 模块的头文件包含路径；
- ngx_module_deps : 模块依赖的头文件；
- ngx_module_libs : 模块需要的链接库；
- ngx_module_order : 仅用于动态 filter 模块，定义模块的顺序。

在脚本里写文件路径时还可以使用变量 \$ngx_addon_dir，它就是命令行 “--add_module=PATH” 里的 PATH，也就是模块的存放路径。

5.3.3 模块脚本

在设置完模块变量后我们必须执行 “auto/module” 脚本，也就是：

```

. auto/module                                #调用 Nginx 提供的脚本处理模块变量

```

脚本 auto/module 将使用这些变量产生 Makefile 和 objs/nginx_modules.c 里的 ngx_modules 模块数组。

在编写模块的 config 脚本时千万不要忘记调用“auto/module”脚本，否则模块就不能集成进 Nginx。

5.3.4 两种脚本格式

目前 Nginx 的编译脚本是在 1.9.11 之后改用的新格式 (New Style)，相应的，之前的编译脚本就称为“旧格式” (Old Style)。

Nginx 提供变量 `$ngx_module_link` 供模块的 config 脚本使用，它的含义是：

- 变量不存在：模块被旧版本 Nginx 编译，不能使用新格式，只能是静态链接；
- ADDON：模块被新版本 Nginx 编译，可以使用新格式，要求静态链接；
- DYNAMIC：模块被新版本 Nginx 编译，可以使用新格式，要求动态链接。

这样，通过 Shell 编程，模块就能够同时兼容新旧版本的 Nginx 编译，还可以针对静态链接和动态链接做出不同的处理，例如：

```
if test -n "$ngx_module_link"; then          #检查$ngx_module_link 是否存在
    ngx_module_type=HTTP
    ...                                     #使用新格式的编译脚本
else                                         #不存在，是旧版本 Nginx
    ...                                     #使用旧格式的编译脚本
fi
```

5.3.5 旧式编译脚本

旧格式的 config 里仍然需要使用 `ngx_addon_name` 设置模块的名字，但不能使用 `ngx_module_type` 等变量，而要用以下的“内部”变量：^①

- NGX_ADDON_SRCS：模块的源码文件，可以有多个；
- HTTP_MODULES：标准的 http 模块用此变量设置模块名字；
- HTTP_FILTER_MODULES：http 过滤模块用此变量设置模块名字；
- STREAM_MODULES：流 (TCP/UDP) 模块用此变量设置模块名字；
- CFLAGS：设置特殊的编译选项；
- HTTP_INCS：编译 http 模块需要包含的路径；
- CORE_INCS：编译所有模块都需要包含的路径；

^① 在 auto/modules 脚本里可以看到还有 `CORE_MODULE`、`HTTP_HEADERS_FILTER_MODULE` 等变量，读者可以自行研究。

- HTTP_DEPS : 编译 http 模块依赖的文件;
- CORE_DEPS : 编译所有模块都依赖的文件;
- NGX_ADDON_DEPS : 仅模块自己依赖的文件;
- CORE_LIBS : 编译时需要链接的外部库。

注意这些变量的赋值都要使用如下方式:

```
VAR="$VAR xxx_module" #注意变量的赋值方式
```

例如:

```
HTTP_MODULES="$HTTP_MODULES ndg_test_module"  
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ModNdgTest.cpp"
```

这种方式相当于把新的字符串附加到了变量的末尾, 相当于 C/C++ 里的 “+=” 操作符。

5.4 总结

本章简要介绍了如何开发 Nginx 模块, 并完整地讲解了 Nginx 的编译脚本知识。

我们首先编写了一个 Nginx 模块 `ndg_test_module`, 它的功能虽然简单, 但却很利于学习, 包含了 Nginx 模块开发的所有基本要素, 也走过了所有开发流程, 完全可以藉此为起点, 实现出更复杂的模块。

以 `ndg_test_module` 为例, 我们介绍了 Nginx 模块开发的基本流程, 包括设计、开发、编译、测试验证和调优五个步骤。其中前四个步骤是开发任何模块所必需的, 而最后一个调优步骤因模块而异, 小且简单的模块通常可以省略, 但大型模块则必不可少, 而且调优所花费的时间可能会比开发和测试更多。

Nginx 使用 Shell 编程, 用脚本的方式产生用于编译的 Makefile 等文件, 是模块的一个重要组成部分, 决定了模块以何种方式集成进 Nginx。

`auto/modules` 脚本会调用模块的 `config` 脚本, 此时 `config` 脚本可以操作 `ngx_module_name`、`ngx_module_type`、`ngx_module_srcs` 等 Shell 变量, Nginx 用这些变量收集第三方模块的源文件、模块变量名、包含路径等信息, 最后由 `auto/make` 脚本生成 Makefile 等文件。

Nginx 允许模块编译为动态模块或静态模块, `config` 脚本需要用变量 `$ngx_module_link` 做逻辑判断, 以兼容旧版本的 Nginx。

目前 Nginx 里最常用、最成熟的是 http 模块, 所以接下来的几章我们将主要围绕 http 模块来详细讲解更多的 Nginx 知识。

第 6 章

Nginx 模块体系

第 5 章初步介绍了 Nginx 开发的基本知识，本章主要讲解 Nginx 的模块体系，即模块架构和配置，包括结构体定义、配置文件解析和相关的操作函数等更深入的细节，最后以 C++ 实现对它们的封装，开发出一个完整的 C++ Nginx 模块。

6.1 模块架构

模块化架构是 Nginx 框架的核心，整个 Nginx 就是由各种 core、event、http、stream 等模块搭建起来的。早期（1.9.11 之前）Nginx 使用的是静态模块，近两年又增加了动态模块，本节我们将详细研究模块的定义、种类、组织形式、加载方式等内容。

6.1.1 结构定义

Nginx 的模块使用 `ngx_module_t` 结构体描述，它的成员很多，但有的只是暂时无用的保留字段，我们并不需要关心。

`ngx_module_t` 定义在 `<core/ngx_module.h>` 里，代码如下：^①

```
// 定义在 core/ngx_core.h
typedef struct ngx_module_s      ngx_module_t;

// 定义在 core/ngx_module.h
struct ngx_module_s {
```

^① 在 Nginx 1.9.11 引入动态模块之前，`ngx_module_t` 位于 `<core/ngx_conf_file.h>`，并且结构定义也与现在的不同，没有 `name`、`signature` 等字段。

```

    ngx_uint_t      ctx_index;    //在 type 类模块数组里的序号
    ngx_uint_t      index;        //在所有模块数组里的序号

    char*           name;         //模块的名字, 标准 C 字符串

    ngx_uint_t      spare0;       //保留字段, 暂无意义
    ngx_uint_t      spare1;       //早期版本有 4 个 spareN 字段

    ngx_uint_t      version;      //模块的版本, 值是 nginx_version
    const char*     signature;    //模块的“签名”、“特征”信息

    void*           ctx;          //特定模块的专用数据, 通常是函数指针表
    ngx_command_t*  commands;     //模块的指令数组, 定义在配置文件里的指令
    ngx_uint_t      type;         //模块的类型标记

    //进程、线程的初始化和退出时的回调函数指针, 共 7 个
    ngx_int_t      (*init_master)(ngx_log_t *log);           //暂无用
    ngx_int_t      (*init_module)(ngx_cycle_t *cycle);
    ngx_int_t      (*init_process)(ngx_cycle_t *cycle);
    ngx_int_t      (*init_thread)(ngx_cycle_t *cycle);       //暂无用
    void           (*exit_thread)(ngx_cycle_t *cycle);       //暂无用
    void           (*exit_process)(ngx_cycle_t *cycle);
    void           (*exit_master)(ngx_cycle_t *cycle);

    uintptr_t      spare_hook0;   //保留字段, 共 8 个, 暂无意义
    ...
    uintptr_t      spare_hook7;
};

```

成员 `name` 是一个普通的 C 字符串, 它标记了模块的名字。

成员 `version` 指示当前 Nginx 的版本, 固定取值为宏 `nginx_version` (定义在 `core/nginx.h`), 对于 Nginx 1.12 来说就是整数 1012000。

成员 `commands` 是一个 `ngx_command_t` 数组指针, 保存了本模块使用的所有配置指令信息, Nginx 会使用它查找指令是否可被当前模块处理。

`init_master` 等函数指针相当于 C++ 里的类成员函数, Nginx 框架将在适当的时机回调这些函数, 完成模块特定的功能。但大多数情况下我们并不需要在模块的初始化和退出时执行动作, 可以直接把它们置为空指针。^①

^① 目前 `init_master`、`init_thread` 和 `exit_thread` 这三个函数并没有被框架调用, 所以模块只能使用 `init_module` 等四个函数指针。

ngx_module_t 里名为 spareN 和 spare_hookN 的成员都是 Nginx 的保留字段, 在目前的代码中并没有使用。

为了简化设置, Nginx 定义了宏 NGX_MODULE_V1_PADDING, 用于初始化 ngx_module_t 的后 8 个成员:

```
#define NGX_MODULE_V1_PADDING    0, 0, 0, 0, 0, 0, 0, 0
```

剩下的 ctx_index、index、signature、type、ctx 等成员关系到 ngx_module_t 的重要功能, 请读者继续阅读。

6.1.2 模块的签名

早期的 Nginx 只支持静态模块, 模块都是以静态链接的方式与 Nginx 框架代码集成在二进制可执行文件里。在 1.9.11 版本之后, Nginx 引入了动态模块, 允许把模块编译成动态库, 用指令 “load_moudle” 在运行时动态加载。

动态模块和 Nginx 主程序可以分离编译, 各自独立发布, 虽然增加了 Nginx 的灵活性, 但也带来了新的问题。其中一个关键的问题就是: 如何保证模块的 “兼容性”, 也就是如何判断一个来源不明的动态模块能否被当前 Nginx 主程序正确加载并调用运行。

Nginx 对此给出的答案是使用 “签名”, 依据操作系统和编译环境的各种特征生成一个 “签名” 字符串, 作为模块的 “特征码”, 只有符合 “签名” 的模块才能被主程序加载。^①

模块 “签名” 的定义摘要如下:

```
// 定义在 core/nginx_module.h
#define NGX_MODULE_SIGNATURE_0          \           //第 0 个特征点
    ngx_value(NGX_PTR_SIZE) " "          \           //指针的大小, 64 位是 8 字节
    ngx_value(NGX_SIG_ATOMIC_T_SIZE) " " \           //sig_atomic_t 结构的大小
    ngx_value(NGX_TIME_T_SIZE) " "       \           //time_t 结构的大小

#if (NGX_HAVE_KQUEUE)
#define NGX_MODULE_SIGNATURE_1 "1"      //kqueue 调用, 也就是 FreeBSD 系统
#else
#define NGX_MODULE_SIGNATURE_1 "0"
#endif

...                                     //其他 epoll、fastopen、reuseport、pcre 等的判断
```

^① 虽然 signature 通常翻译为 “签名”, 但这里理解成 “特征码” 可能更合适一些。

```
#define NGX_MODULE_SIGNATURE \
    NGX_MODULE_SIGNATURE_0 NGX_MODULE_SIGNATURE_1 \
    ...
    NGX_MODULE_SIGNATURE_33 NGX_MODULE_SIGNATURE_34 //总共有 35 个特征值
```

通过这种方式, Nginx 把硬件体系、编译环境、操作系统、底层调用等各种信息转换成了一个特征字符串, 主程序和任何一个模块都持有这个“签名”, 在动态加载模块时就比对这个“签名”——如果签名一致就表明模块与当前的 Nginx 主程序是“兼容的”, 各种运行参数都匹配, 可以正确加载, 反之则意味着模块有一项或多项参数不匹配, 不能加载。

例如, 当前的 Nginx 是在 64 位的 Linux 上编译的, 而有一个动态模块是在 32 位的 FreeBSD 上编译的, 那么因为 `NGX_MODULE_SIGNATURE_0` 和 `NGX_MODULE_SIGNATURE_1` 这两个特征点不同, 模块就无法加载, 从而保护了 Nginx 主程序。

`ngx_module_t` 里的成员 `signature` 就是模块里存储“签名”的位置。

Nginx 还定义了填充宏 `NGX_MODULE_V1` 简化签名的设置, 它不仅可以初始化签名, 还可以一并用于初始化 `ngx_module_t` 里的前 7 个成员:^①

```
#define NGX_MODULE_V1 \
    NGX_MODULE_UNSET_INDEX, NGX_MODULE_UNSET_INDEX, \
    NULL, 0, 0, ngx_version, NGX_MODULE_SIGNATURE
```

注意在宏 `NGX_MODULE_V1` 里, 前两个 `ctx_index` 和 `index` 字段被填充为“-1”, 也就是说序号“未初始化”。

6.1.3 模块的种类

Nginx 框架定义了 6 种类型的模块, 分别是 `core`、`conf`、`event`、`stream`、`http` 和 `mail`, 所有的 Nginx 模块都必须属于这 6 类模块。

`ngx_module_t::type` 表示模块的类型, 取值必须是以下 6 个宏:

```
#define NGX_CORE_MODULE      0x45524F43 // "CORE"
#define NGX_CONF_MODULE      0x464E4F43 // "CONF"
#define NGX_EVENT_MODULE     0x544E5645 // "EVNT"
#define NGX_STREAM_MODULE    0x4d525453 // "STRM"
#define NGX_HTTP_MODULE      0x50545448 // "HTTP"
#define NGX_MAIL_MODULE      0x4C49414D // "MAIL"
```

本书中开发的模块大多数是 HTTP 模块, 所以使用宏 `NGX_HTTP_MODULE`。

^① 在 1.9.11 之前的 Nginx 源码里, 宏 `NGX_MODULE_V1` 是简单的“0,0,0,0,0,0,1”。

6.1.4 模块的函数指针表

ngx_module_t 里的成员 type 相当于类型的标记 (RTTI)，而 ctx 则是一个函数指针表，类似 C++ 里的虚函数表，不同的模块可以有不同的定义，实现了模块的“子类化”——这是因为 C 语言没有 C++ 的继承机制。

ctx 类型是 void*，意味着它的内容是不确定的，必须结合 type 才能确定 ctx 的具体含义。

Nginx 的 6 类模块都定义了各自的 ctx 结构，名字是 ngx_xxx_module_t。^①

core 模块

core 模块的 ctx 是：

```
typedef struct {
    ngx_str_t      name;
    void           (*create_conf)(ngx_cycle_t *cycle);
    char           (*init_conf)(ngx_cycle_t *cycle, void *conf);
} ngx_core_module_t;
```

ngx_core_module_t 结构比较简单，只有两个函数指针，仅用于创建和初始化配置结构体，这是因为 core 模块通常不处理具体的业务，只负责构建子系统。

目前 Nginx 一共只有为数不多的几个 core 模块，包括 ngx_core_module、ngx_errlog_module、ngx_events_module、ngx_http_module、ngx_stream_module 等，它们是 Nginx 框架里最底层最核心的模块，Nginx 框架直接与这些核心模块交互，而 ngx_events_module 和 ngx_http_module 这样的业务基础模块则再自行构建出自己特定的模块体系和运行机制。

对于 ngx_http_module 来说，它只负责把 http 模块管理组织起来，接入 Nginx 框架，真正的 HTTP 业务逻辑则由 HTTP 核心模块 ngx_http_core_module 来实现。

http 模块

http 模块是我们最常用的模块，它的 ctx 类型是 ngx_http_module_t，定义如下：

```
//定义在 http/ngx_http_config.h
typedef struct {
    ngx_int_t      (*preconfiguration)(ngx_conf_t *cf);
    ngx_int_t      (*postconfiguration)(ngx_conf_t *cf);
```

^① ngx_conf_module 比较特殊，它没有 ctx 扩展，所以 ctx==nullptr。

```

void*      (*create_main_conf)(ngx_conf_t *cf);
char*      (*init_main_conf)(ngx_conf_t *cf, void *conf);

void*      (*create_srv_conf)(ngx_conf_t *cf);
char*      (*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);

void*      (*create_loc_conf)(ngx_conf_t *cf);
char*      (*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
} ngx_http_module_t;

```

可以看到, `ngx_http_module_t` 结构定义了 8 个函数指针, 它们都是配置解析相关的函数, 在 Nginx 框架解析配置文件时被调用, 具体的回调时机是:

- `preconfiguration` : 开始解析 http 块前调用, 常用于添加变量;
- `postconfiguration` : 解析完 http 块后调用, 常用于初始化模块 handler;
- `create_main_conf` : 创建 http main 级别的配置数据结构;
- `init_main_conf` : 初始化 http main 级别的配置数据结构;
- `create_srv_conf` : 创建 server 级别的配置数据结构;
- `merge_srv_conf` : 合并 main 级别和 server 级别的配置数据结构;
- `create_loc_conf` : 创建 location 级别的配置数据结构;
- `merge_loc_conf` : 合并 server 级别和 location 级别的配置数据结构。

这些函数的调用与配置文件的解析密切相关, 需要使用 `ngx_conf_t` 结构在 Nginx 的启动阶段设置模块的各种参数, 比较常用的是 `preconfiguration`、`postconfiguration`、`create_loc_conf` 和 `merge_loc_conf` 这几个函数, 我们会在随后进一步介绍。

6.1.5 模块的类图

如果从 C++ 的视角来观察 Nginx 的模块架构, 那么 `ngx_module_t` 就是一个抽象基类, `ngx_core_module_t`、`ngx_http_module_t` 以 `ctx` 的方式继承了 `ngx_module_t`, 最后的各个具体模块实现了类里的函数指针, 是它们的实现类。

使用 UML 可以画出 Nginx 模块的示意类图, 对于我们理解 Nginx 模块架构很有参考意义, 如图 6-1 所示。

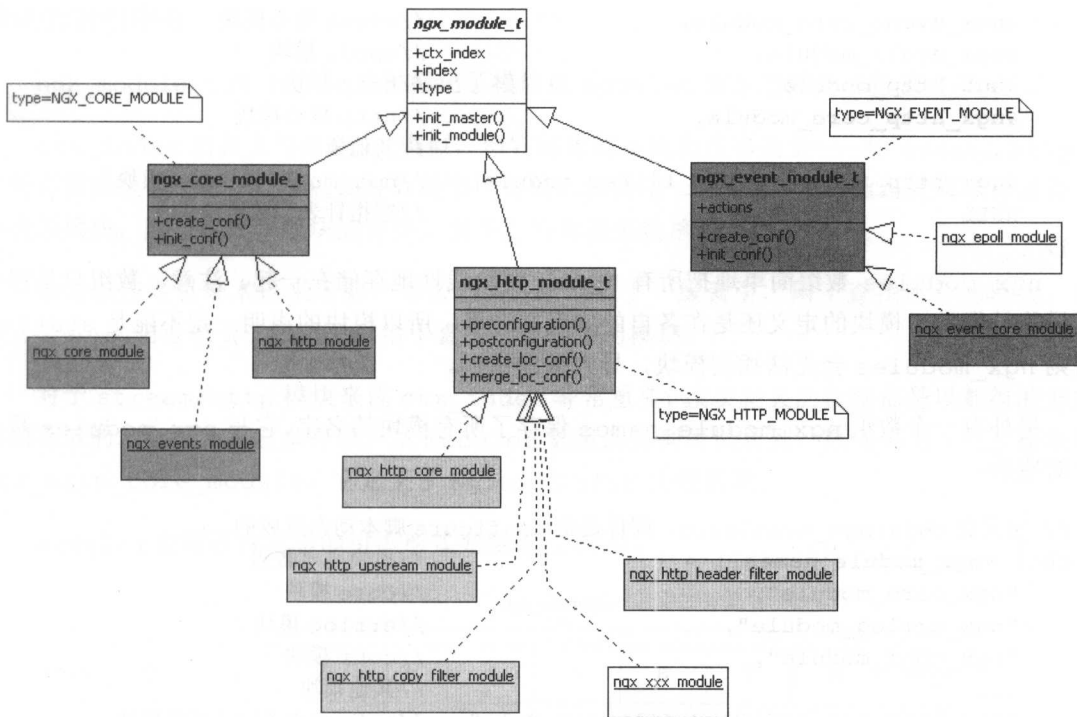


图 6-1 Nginx 模块的类图

6.1.6 模块的组织形式

Nginx 包含了数十个功能各异的模块，之间的层次和调用关系非常复杂，如果不阅读 Nginx 源码，可能有的人会以为众多的模块是以某种复杂的数据结构组织起来的，但实际情况完全相反，Nginx 并没有使用自定义的那些高级数据结构，而只是使用了最简单的 C 内建数组。

ngx_modules.c

Nginx 的 `configure` 脚本会生成源码文件 `objs/ngx_modules.c`，在这里所有的静态 Nginx 模块都被放进了一个普通的数组，用于启动时的初始化，例如：

```
// 定义在 objs/ngx_modules.c, 是由 configure 脚本动态生成的
ngx_module_t *ngx_modules[] = {
    &ngx_core_module,           //core 模块
    &ngx_errlog_module,         //errlog 模块
    &ngx_conf_module,           //conf 模块
    &ngx_regex_module,          //regex 模块
    &ngx_events_module,         //event 模块
    // ... other modules ...
};
```

```

    &ngx_event_core_module,           //event 核心模块
    &ngx_epoll_module,                //epoll 模块
    &ngx_http_module,                  //http 模块
    &ngx_http_core_module,             //http 核心模块
    ...                               //其他模块
    &ngx_http_not_modified_filter_module, //not_modified 过滤模块
    NULL                             //空指针表示数组结束
};

```

ngx_modules 数组简单地把所有 Nginx 模块线性地存储在一起。注意，数组只是保存了模块的指针，模块的定义还是在各自的实现文件里，所以模块的声明一定不能是 static，否则 ngx_modules 会无法找到模块，导致编译错误。

另外有一个数组 ngx_module_names 保存了所有模块的名字，它与 ngx_modules 是一一对应的：

```

// 定义在 objs/nginx_modules.c，同样是由 configure 脚本动态生成的
char *ngx_module_names[] = {           //模块名字数组
    "ngx_core_module",                  //core 模块
    "ngx_errlog_module",                //errlog 模块
    "ngx_conf_module",                  //conf 模块
    ...                                 //其他模块
    "ngx_http_not_modified_filter_module", //not_modified 过滤模块
    NULL                               //空指针表示数组结束
};

```

ngx_cycle_t

ngx_cycle_t 是 Nginx 框架的核心数据结构，是 Nginx 运行的整个“生命周期”（cycle）都要使用到的数据，它里面有一个数组成员 modules，保存了运行时的所有模块——既有编译时确定的静态模块，也有运行时加载的动态模块：

```

// 定义在 core/nginx_core.h
typedef struct ngx_cycle_s              ngx_cycle_t;

// 定义在 core/nginx_cycle.h
struct ngx_cycle_s {
    ...
    ngx_module_t**      modules;          //运行时模块指针数组，所有的模块都在这里
    ngx_uint_t           modules_n;        //模块数组里可用的序号
    ngx_uint_t           modules_used;     //模块是否已经加载的标志位
};

```

modules 数组仅仅起到集中所有模块的作用，为 Nginx 框架提供一个访问所有模块的入口，模块的层次和调用关系则由模块数据结构里的 ctx_index、index 和各种指针来实现。

模块的索引序号

ngx_module_t 的 index 成员标记了模块在 modules 数组里的索引位置（序号）。
ctx_index 的含义与 index 类似，但它标记的是模块在本类型——如 event、http——所有模块里的序号。这样做的好处是节约存储，只需用一个 modules 数组就可以存放各个种类的模块，间接实现了模块的分类，而不必为每类模块单独建立数组。

如果以搜索引擎术语来理解，那么 index 近似于一级索引，用于查找所有的模块，而 ctx_index 则近似于二级索引，用于查找特定类型的模块。

对于 stream/http 模块来说 ctx_index 非常重要，模块相关的数据都是以数组的方式依次存放的，必须使用它来存取。从 modules 数组的排列可以看到，第 0 号的 http 模块是 ngx_http_core_module，它定义了 Nginx 的 HTTP 处理框架。

modules 数组在内存里的组织形式如图 6-2 所示。

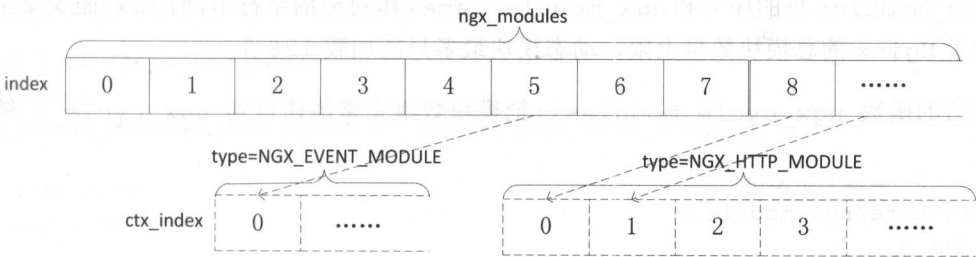


图 6-2 ngx_modules 数组的内存组织形式

6.1.7 模块的初始化

index 和 ctx_index 通常都使用宏 NGX_MODULE_V1 设置，在最初都是未初始化的（即“-1”），Nginx 会在启动阶段根据模块数组进行合适的设置（可参考第 13 章）。

index 的初始化

Nginx 启动时，main()函数调用 ngx_preinit_modules()顺序遍历 ngx_modules 数组，设置每个静态模块的 index 成员，代码摘要如下：

```
// 位于 core/nginx_module.c
#define NGX_MAX_DYNAMIC_MODULES 128 //动态模块最多加载 128 个

ngx_uint_t      ngx_max_module;      //模块数量的上限
static ngx_uint_t ngx_modules_n;      //模块序号的计数器
```

```

ngx_int_t
ngx_preinit_modules(void)                                //初始化模块数组
{
    ngx_uint_t i;

    for (i = 0; ngx_modules[i]; i++) {                  //遍历 ngx_modules 数组
        ngx_modules[i]->index = i;                      //设置模块的 index 成员
        ngx_modules[i]->name = ngx_module_names[i];     //设置模块的名字
    }

    ngx_modules_n = i;                                   //数组的最后一个可用序号
    ngx_max_module =                                   //模块数量的上限
        ngx_modules_n + NGX_MAX_DYNAMIC_MODULES;        //静态模块数量加上 128

    return NGX_OK;
}

```

ngx_preinit_modules() 执行之后, 每个静态模块的 index 和 name 就确定了, 分别是在 ngx_modules 里的序号和 ngx_module_names 里对应的字符串, 而 ngx_max_module 则确定了 Nginx 的总模块数量上限, 动态模块最多只能加载 128 个。

随后的函数 ngx_cycle_modules() 把模块数组完整地拷贝到 ngx_cycle_t 结构体里:

```

// 位于 core/nginx_module.c
ngx_int_t
ngx_cycle_modules(ngx_cycle_t *cycle)
{
    cycle->modules = ngx_palloc(                          //使用内存池分配内存
        cycle->pool, (ngx_max_module + 1) * sizeof(ngx_module_t *));

    ngx_memcpy(cycle->modules, ngx_modules,                //拷贝数组
        ngx_modules_n * sizeof(ngx_module_t *));

    cycle->modules_n = ngx_modules_n;                      //拷贝数组可用序号

    return NGX_OK;
}

```

拷贝完成之后, 所有静态模块的加载就完成了, ngx_modules、ngx_module_names 和 ngx_modules_n 这三个全局变量也就完成了“历史使命”, Nginx 在之后的生命周期里都不再使用。

ctx_index 的初始化

以 http 模块为例，在 ngx_http_module（它是 core 模块而不是 http 模块）里，当调用 ngx_http_block() 函数解析配置文件 http 块时，Nginx 会调用 ngx_count_modules() 函数，初始化 http 模块的 ctx_index 成员：

```
// 位于 http/ngx_http.c
ngx_uint_t    ngx_http_max_module;           //http 模块的总数，不是上限
```

```
static char *
ngx_http_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    ...
    ngx_http_max_module = ngx_count_modules(cf->cycle, NGX_HTTP_MODULE);
    ...
}
```

函数 ngx_count_modules() 遍历 cycle->modules 数组，检查模块的 type 成员，为模块设置正确的 ctx_index，代码摘要如下：

```
// 位于 core/ngx_module.c
ngx_int_t
ngx_count_modules(ngx_cycle_t *cycle, ngx_uint_t type)
{
    next = 0;                                //初始化计数器
    max = 0;

    for (i = 0; cycle->modules[i]; i++) {     //遍历模块数组
        module = cycle->modules[i];           //拿到数组里的一个模块

        if (module->type != type) {           //检查模块类型
            continue;                         //不是指定的类型则不处理
        }

        module->ctx_index =                   //调用 ngx_module_ctx_index() 获取一个序号
            ngx_module_ctx_index(cycle, type, next);

        if (module->ctx_index > max) {         //更新 max
            max = module->ctx_index;
        }
    }
    cycle->modules_used = 1;                  //已经完成模块的初始化，不能再添加模块

    return max + 1;                          //返回此类型模块的总数
}
```

函数最后返回 `max+1` 是为了给数组最后位置存放空指针而留的，标记数组结束。

6.1.8 模块的动态加载

当使用编译选项“`--add-dynamic-module`”时，Nginx 会自动生成一个编译动态模块专用的同名 C 源码文件，里面定义了三个数组，形式与 `ngx_modules.c` 很类似，记录了模块的基本信息，例如：

```
ngx_module_t *ngx_modules[] = {           //动态库里的所有模块
    &xxx_module,
    NULL
};

char *ngx_module_names[] = {              //动态库里的模块对应的名字
    "xxx_module",
    NULL
};

char *ngx_module_order[] = {              //动态库里的模块的顺序关系，用于过滤链
    NULL
};
```

这三个数组是全局的，所以在“`make -shared`”编译成动态库时就是可加载的符号（可以用命令“`nm -D`”查看）。

Nginx 在启动过程中解析配置文件遇到“`load_module`”指令时，就会调用函数 `ngx_load_module()`，利用这三个数组尝试加载动态模块，代码摘要如下：

```
static char *
ngx_load_module(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    if (cf->cycle->modules_used) {         //已经完成模块的初始化，不能再添加模块
        return "is specified too late";
    }

    value = cf->args->elts;                 //取指令数组
    file = value[1];                       //第 1 个元素就是模块的文件名

    if (ngx_conf_full_name(cf->cycle, &file, 0) != NGX_OK) { //计算正确的路径
        return NGX_CONF_ERROR;
    }

    handle = ngx_dlopen(file.data);        //打开动态库
```



```

modules = ngx_dlsym(handle, "ngx_modules");           //取动态库里的模块数组
names = ngx_dlsym(handle, "ngx_module_names");        //取动态库里的名字数组
order = ngx_dlsym(handle, "ngx_module_order");        //取动态库里的顺序数组

for (i = 0; modules[i]; i++) {                        //遍历动态库里的模块数组
    module = modules[i];                             //拿到一个动态模块
    module->name = names[i];                          //设置模块的名字

    //调用 ngx_add_module 把模块加入 cycle->modules 数组
    if (ngx_add_module(cf, &file, module, order) != NGX_OK) {
        return NGX_CONF_ERROR;
    }
}

return NGX_CONF_OK;                                  //完成动态模块的加载
}

```

函数首先检查 `cycle->modules_used` 标志位, 如果为 `true`, 那么就意味着之前已经有 `events/stream/http` 等模块调用了 `ngx_count_modules()`, 执行了模块的初始化, 这时再加载动态模块就是不可能的了。

之后的操作比较容易理解, 使用 `dlopen/dlsym` 等系统调用打开动态库并获取里面的数组, 数组里存放的是模块相关的定义, 执行 `ngx_add_module()` 加入 `cycle->modules` 数组。

`ngx_add_module()` 函数检查版本号、签名和其他信息, 最终完成模块的加载:

```

ngx_int_t
ngx_add_module(...)
{
    if (cf->cycle->modules_n >= ngx_max_module) {     //超过模块数量上限
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
            "too many modules loaded");
        return NGX_ERROR;
    }

    if (module->version != nginx_version) {           //模块的版本号不匹配
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
            "module \"%V\" version %ui instead of %ui",
            file, module->version, (ngx_uint_t) nginx_version);
        return NGX_ERROR;
    }

    if (ngx_strcmp(                                  //模块的签名不匹配
        module->signature, NGX_MODULE_SIGNATURE) != 0) {
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,

```

```

        "module \"%V\" is not binary compatible",
        file);
    return NGX_ERROR;
}

if (module->index == NGX_MODULE_UNSET_INDEX) {    //模块的序号还未设置
    module->index = ngx_module_index(cf->cycle); //获取一个序号

    if (module->index >= ngx_max_module) {        //超过模块数量上限
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
            "too many modules loaded");
        return NGX_ERROR;
    }
}

... //计算模块的正确插入位置

cf->cycle->modules[before] = module; //把模块加入数组
return NGX_OK;
}

```

6.2 配置解析

Nginx 的模块化架构是与灵活的配置文件格式紧密联系在一起的，模块决定了配置文件的结构和指令，配置文件再用这些指令来调整模块的行为，两者相辅相成，要想完全掌握 Nginx 的模块架构和运行机制，就必须深刻理解 Nginx 的配置解析功能。

本节将从 `ngx_cycle_t`、`ngx_http_conf_ctx_t`、`ngx_conf_t` 和 `ngx_command_t` 这几个关键数据结构入手，详细剖析 Nginx 的配置解析机制。

6.2.1 结构定义

本节简要介绍数据结构 `ngx_cycle_t`、`ngx_http_conf_ctx_t`、`ngx_conf_t` 和 `ngx_command_t`，它们代表了 Nginx 配置数据的静态存储。

`ngx_cycle_t`

`ngx_cycle_t` 不仅存储了所有的模块信息，也存储了所有的配置解析相关信息：

```

// 定义在 core/ngx_cycle.h
struct ngx_cycle_s {
    ngx_module_t**    modules;    //运行时模块指针数组，所有的模块都在这里

```

```

void****          conf_ctx;          //配置数据的起始存储位置
...              //其他数据成员
};

```

conf_ctx 是 Nginx 存储配置数据的起点，但它的声明“非常可怕”，使用了连续四个星号，任何人都很难立即理解其含义。

我们可以使用 typedef 来简化代码，转化为较为容易理解的形式：

```

typedef void**      void_array;        //即 void*[], 存储 void* 的数组
typedef void_array** void_multi_array; //即 void_array*[]
void_multi_array    conf_ctx;          //定义 conf_ctx

```

这样就可以比较清晰地看到，conf_ctx 实际上是一个二维数组，数组里的元素是 void* 指针，每一个指针又指向了另一个存储 void* 的数组（实际对应到 event 模块的配置存储）。

不过由于 void* 指针的易变性，conf_ctx 还可以根据模块的具体情况转换为 void** 类型，这时它表示的是一个普通数组，里面存储的是 void* 指针（允许任意解释）。

在 ngx_cycle.c 的 ngx_init_cycle() 函数里创建 conf_ctx 的代码是：

```

cycle->conf_ctx = ngx_pcalloc(pool, ngx_max_module * sizeof(void *));

```

这实际上就是一个存储 void* 的普通数组，长度为 ngx_max_module，可以用来存放所有模块的配置数据结构（但实际上 Nginx 并没有这么做）。

Nginx 通过这种方式灵活地支持了不同模块的数据存储需求：对于 core 模块指针直接指向配置数据，而 event/stream/http 模块则指针指向的可以是另外一个数组，形成了一个树形结构。

ngx_http_conf_ctx_t

ngx_http_module 是 core 模块，它的配置数据结构是 ngx_http_conf_ctx_t，用来存储 http 模块的配置数据：

```

// 定义在 http/ngx_http_config.h
typedef struct {
    void**      main_conf;          //http main 域的存储数组
    void**      srv_conf;           //server 域的存储数组
    void**      loc_conf;          //location 域的存储数组
} ngx_http_conf_ctx_t;

```

这里的主 main_conf、srv_conf 和 loc_conf 与 conf_ctx 的功能相似，也是 void* 数组。但它们使用的序号是 ctx_index，只存放所有 http 模块 create_xxx_conf 创建的数据，其他 event、stream 等模块则不在其中。

使用三个数组的原因是 Nginx 把 http 配置设计为 http/server/location 三个层次，每个模块可以为不同的层次使用不同的配置，所以必须分别存储。

ngx_conf_t

ngx_conf_t 结构定义在 ngx_conf_file.h，是 Nginx 在解析配置文件时的重要数据结构，表示解析当前配置指令时的运行环境数据 (Context)：^①

```
// 定义在 core/nginx_core.h
typedef struct ngx_conf_s      ngx_conf_t;

// 定义在 core/nginx_conf_file.h
struct ngx_conf_s {
    ngx_array_t*                args;                // 配置文件里的指令字符串数组

    ngx_pool_t*                 pool;                 // 内存池对象
    ngx_log_t*                   log;                 // 日志对象

    void*                        ctx;                 // 当前环境，例如 ngx_http_conf_ctx_t
    ngx_uint_t                   cmd_type;             // 当前的命令类型

    ...                          // 其他数据成员
};
```

因为 Nginx 的配置文件是有层次的，分成了 main/http/server/location 等作用域，不同的域里指令的含义和处理方式都有可能不同（例如 http 块里的 server 指令和 upstream 块里的 server 指令就是完全不同的），所以 Nginx 在解析配置文件时必须使用 ngx_conf_t 来保存当前的基本信息，进入退出一个配置块都会变更 ngx_conf_t，指令的解析必须要参考 ngx_conf_t 环境数据才能正确处理。

args 参数是我们最常用到的成员，它是一个 ngx_array_t，以 ngx_str_t 的形式存储了分割好的配置文件指令字符串。如果有 ngx_str_array arr(cf->args)，那么 arr[0] 就是指令名，arr[1] 就是第一个参数，以此类推。

pool 成员是内存池，可以使用它分配内存，通常是创建配置数据结构，我们已经在 5.1.2

^① Context 是程序开发时经常使用的术语，它的出现意味着代码的运行不是完全独立的，代码的行为可能会依据 Context 而发生改变，作用相当于一个全局变量的集合。Context 直译是“上下文”，但这个名词在作者看来其实是一种“偷懒”的行为，直接照搬了文学领域的翻译，没有精确表达出在计算机编程领域的含义，所以本书中的 Context 中文均写作“运行环境数据”或“环境数据”，表示代码是运行在 Context 这个动态的环境之内。

节看到了它的用法。

log 成员是日志对象，但在配置解析未完成时可能还没有文件可供输出，那么它就会打印到控制台上。

ctx 是 ngx_conf_t 里最灵活的成员，在配置解析过程中随着配置块的切换而发生变化，指示了当前配置解析所需参考的环境数据，对于 http 模块来说 ctx 必定是 ngx_http_conf_ctx_t。

解析 http 指令的 ngx_http_block() 里的相关代码如下：

```
//位于 ngx_http.c
static char *
ngx_http_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    ngx_conf_t      pcf;           //临时保存 ngx_conf_t
    ngx_http_conf_ctx_t* ctx;      //http 块的环境数据

    ctx = ngx_palloc(               //创建环境数据 ngx_http_conf_ctx_t
        cf->pool, sizeof(ngx_http_conf_ctx_t));

    pcf = *cf;                      //暂存之前的 ctx
    cf->ctx = ctx;                   //设置本配置块的 ctx

    cf->module_type = NGX_HTTP_MODULE; //设置模块的类型标志量
    cf->cmd_type = NGX_HTTP_MAIN_CONF; //设置命令的类型标志量

    rv = ngx_conf_parse(cf, NULL);  //使用新的 ctx 解析块内指令

    *cf = pcf;                      //解析完毕，恢复之前的 ctx
    return rv;
}
```

其他 server、location 指令的解析也与此类似，设置本配置块的 ctx，这样块内的所有指令都会以此 ctx 确定正确的含义。

ngx_command_t

ngx_command_t 结构定义在 ngx_conf_file.h，它包含了 Nginx 框架解析指令所需要的全部信息：

```
// 定义在 core/nginx_core.h
typedef struct ngx_command_s      ngx_command_t;
```

```
// 定义在 core/nginx_conf_file.h
struct ngx_command_s {
    ngx_str_t    name;                //指令的名字
    ngx_uint_t   type;                //指令的作用域和类型
    char*        (*set) (              //指令的解析函数
        ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
    ngx_uint_t   conf;                //配置结构所在的存储位置
    ngx_uint_t   offset;              //具体的存储变量的偏移量
    void*        post;                //可以指向任意数据供解析时使用
};
```

ngx_command_t 共有六个成员，它们的含义在注释里已经有了初步解释，下面的小节将做详细讲解。

6.2.2 配置解析的基本流程

完整地阐述 Nginx 的配置解析流程需要结合源码，为节约篇幅这里只做一个比较简略的说明。

Nginx 框架流程

当启动 /usr/local/nginx/sbin/nginx 时，Nginx 从入口函数 main() 开始执行，检查命令行参数和环境变量，然后进入初始化函数 ngx_init_cycle()：

- 1) 创建 cycle->conf_ctx 数组；
- 2) 拷贝 ngx_modules 数组到 cycle->modules；
- 3) 调用所有 core 模块的 create_conf 函数指针，创建出配置结构，填入数组；
- 4) 设置 ngx_conf_t 的各个字段，作为配置解析的起始 ctx；
- 5) 执行 ngx_conf_parse()，在模块数组里查找配置指令对应的解析函数并处理；
- 6) 反复执行步骤 5，直至整个配置文件处理完毕；
- 7) 调用 core 模块 init_conf 函数指针，初始化 core 模块的配置。

可见，Nginx 框架在 ngx_init_cycle() 里只处理 core 模块，它看不到也并不关心 event、http、stream 等模块，各个 core 模块需要实现自己的配置解析功能，这给予了模块极大的自由，core 模块可以灵活定义自己的架构。

core 模块的 init_conf 函数指针需要在流程的最后执行，这是因为只有当解析完整个配置文件后才能确定 core 模块的相关配置信息，如果有的参数没有配置，那么就可以在这个

时候进行初始化，设置默认值。

HTTP 框架流程

`ngx_http_module` 模块定义了 Nginx 的 HTTP 框架，核心函数是解析 http 块的 `ngx_http_block()` 函数，它与 `ngx_init_cycle()` 的功能和流程类似：

- 1) 创建 `ngx_http_conf_ctx_t` 结构，里面有三个配置数组；
- 2) 调用所有 http 模块的 `create_xxx_conf()` 填充数组；
- 3) 设置 `ngx_conf_t` 的各个字段，作为解析 http 模块的基本环境；
- 4) 准备工作，执行每个模块的 `preconfiguration` 函数指针；
- 5) 调用 `ngx_conf_parse()`，解析 http 块内的所有指令，直至解析完毕；
- 6) 调用 `init_main_conf` 函数指针，初始化 main 配置数据；
- 7) 调用所有 http 模块的 `merge_srv_conf` 函数指针，合并 main/srv 域的配置数据；
- 8) 最后执行 `postconfiguration`，完成整个 http 块的配置解析。

server 和 location 块的解析与 http 块类似，区别在于 `ngx_http_conf_ctx_t` 结构里数组元素的填充，可以直接使用上层的数据，无须重复创建，也不需要调用 `preconfiguration`、`postconfiguration` 等函数指针。

6.2.3 配置数据的存储模型

Nginx 的配置文件的块的形式定义了层次关系，分成了 main/http/server/location 四个层次，对应到框架内部的存储也是如此，不同的层次使用不同的内存区域（数组）保存数据，区域之间以指针连接表示层次关系，设计得非常精巧。^①

本节将结合源码和图表来解析 Nginx 配置数据在内存里的存储模型。

main 层次

main 层次的存储使用的是 `ngx_cycle_t::conf_ctx`，它只存储 core 模块的配置数据。

^① 本章暂不讨论 event、stream 等模块，读者可参见第 14 章和第 16 章。

在 ngx_init_cycle() 函数里，conf_ctx 使用 cycle->modules 数组进行初始化，它只操作 core 模块：

```
// 位于 core/nginx_cycle.c
for (i = 0; cycle->modules[i]; i++) {
    if (cycle->modules[i]->type != NGX_CORE_MODULE) { //检查 core 模块
        continue;
    }

    module = cycle->modules[i]->ctx; //获取模块的 ctx 函数表

    if (module->create_conf) { //模块是否有 create 函数
        rv = module->create_conf(cycle); //创建模块的配置数据

        cycle->conf_ctx[cycle->modules[i]->index] = rv; //加入数组
    }
}
```

由于 core 模块数量很少，所以 ngx_cycle_t::conf_ctx 数组几乎是空的，主要存储的是 ngx_core_module 模块的配置数据，用于解析 master、daemon 等指令。^①

main 层次配置数据的存储可以用图 6-3 来表示。

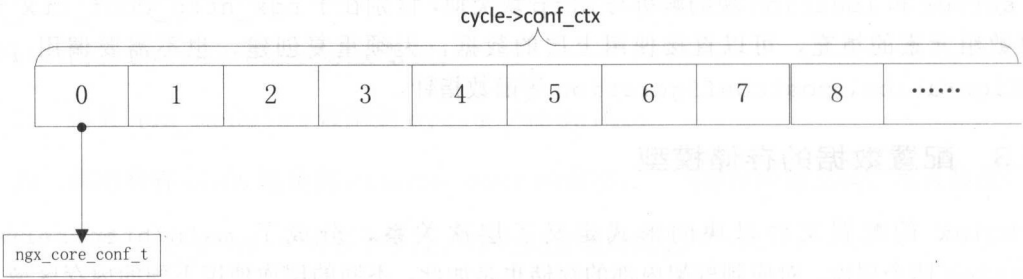


图 6-3 main 层次配置数据的存储

http 层次

当 Nginx 框架遇到 http 指令时，就会调用 ngx_http_block() 函数，在这里会创建 ngx_http_conf_ctx_t 结构，为三个数组分配内存：

```
//位于 http/nginx_http.c, ngx_http_block()
ctx = ngx_palloc(cf->pool, sizeof(ngx_http_conf_ctx_t));
```

^① http、stream 等模块并没有实现 create_conf 函数，只有解析到 http 块或 stream 块时才动态创建配置数据。


```

ctx->main_conf = ngx_palloc(cf->pool,           //分配 main_conf 的内存
                           sizeof(void *) * ngx_http_max_module);
...                                           //为另外两个数组分配内存

for (m = 0; cf->cycle->modules[m]; m++) {
    if (cf->cycle->modules[m]->type != NGX_HTTP_MODULE) { //检查 http 模块
        continue;
    }

    module = cf->cycle->modules[m]->ctx;           //获取模块的 ctx 函数表
    mi = cf->cycle->modules[m]->ctx_index;         //获取模块的类型序号

    if (module->create_main_conf) {               //创建 main 配置
        ctx->main_conf[mi] = module->create_main_conf(cf);
    }

    if (module->create_srv_conf) {                 //创建 server 配置
        ctx->srv_conf[mi] = module->create_srv_conf(cf);
    }

    if (module->create_loc_conf) {                 //创建 location 配置
        ctx->loc_conf[mi] = module->create_loc_conf(cf);
    }
}

```

从这段代码可以看到，http main 层次的 ngx_http_conf_ctx_t 结构不仅保存了所有 http 模块的 main 配置数据，同时也保存了所有 http 模块的 server 和 location 配置数据。这种设计实现了 Nginx 配置文件的指令作用域功能：低层次的配置指令可以在高层次里出现，高层次指令的值作为低层次指令的默认设置，多个低层次的指令可以共享高层次指令的值——只需把两个层次的数据比较再合并即可。

http 层次配置数据的存储可以用图 6-4 来表示。

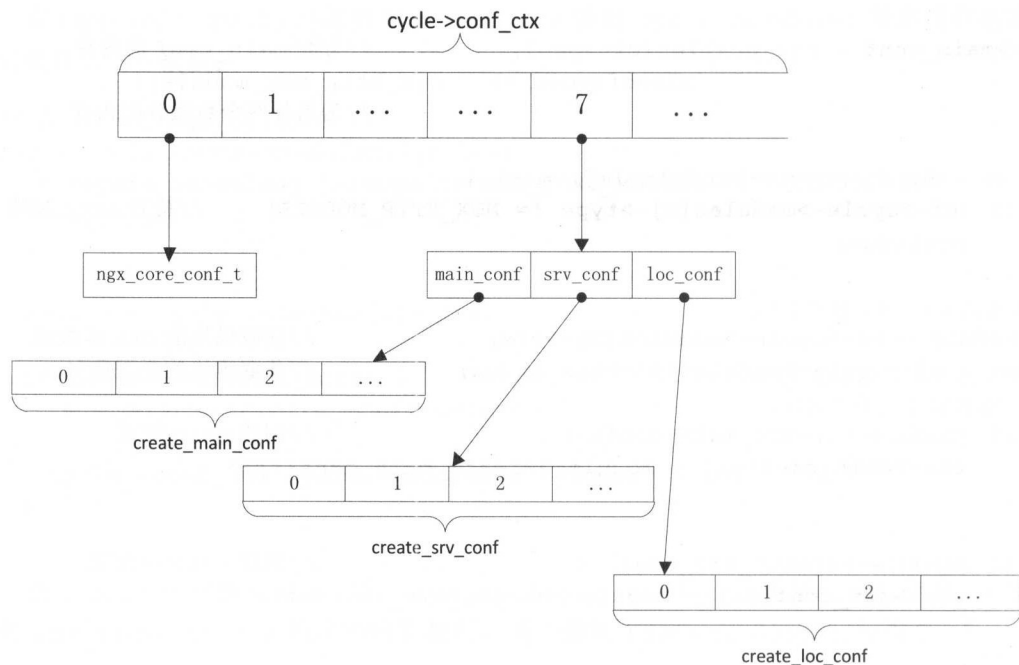


图 6-4 http 层次配置数据的存储

server 层次

`ngx_http_core_module` 模块是第一个 http 模块，它的配置数据结构 `ngx_http_core_main_conf_t` 里有个动态数组成员 `servers`，保存了所有 server 相关的信息（使用结构体 `ngx_http_core_srv_conf_t`）。

每一个 server 块也同样使用 `ngx_http_conf_ctx_t` 里的三个数组来保存模块配置数据，代码与 http 块的处理类似。但因为 http 块的配置指令不会在 server 块里出现，所以不会创建 main 配置数据，而是直接用指针引用：

```
//位于 http/http_core_module.c, ngx_http_core_server()
ctx = ngx_palloc(cf->pool, sizeof(ngx_http_conf_ctx_t));

ctx->main_conf = http_ctx->main_conf;           //指向上层的配置数组

for (i = 0; cf->cycle->modules[i]; i++) {
    if (cf->cycle->modules[i]->type != NGX_HTTP_MODULE) { //检查 http 模块
        continue;
    }

    module = cf->cycle->modules[i]->ctx;           //获取模块的 ctx 函数表
```

```
if (module->create_srv_conf) { //创建 server 配置
    mconf = module->create_srv_conf(cf);

    ctx->srv_conf[cf->cycle->modules[i]->ctx_index] = mconf;
}

if (module->create_loc_conf) { //创建 location 配置
    mconf = module->create_loc_conf(cf);

    ctx->loc_conf[cf->cycle->modules[i]->ctx_index] = mconf;
}
}
```

这样，每个 server 块直接引用了上层 http 块的 main 配置，然后又拥有了只属于本块的 srv 和 loc 配置，既节约了存储成本，又可以简单快速地访问 main 配置，无须跳转。

server 层次配置数据的存储可以用图 6-5 来表示。

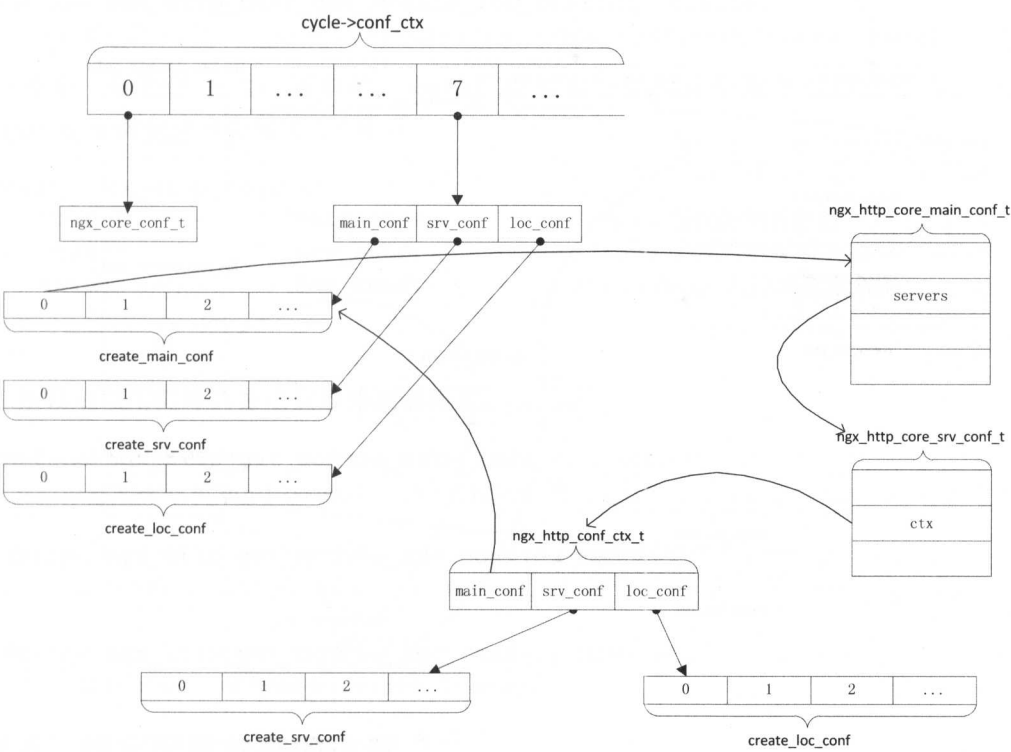


图 6-5 server 层次配置数据的存储

location 层次

ngx_http_core_module 模块使用结构体 ngx_http_core_loc_conf_t 以队列的形式保存 server 里的所有 location 信息。

location 块与 http 和 server 块一样,使用 ngx_http_conf_ctx_t 的三个数组保存模块配置数据,但用指针引用上层 main 和 server 的配置数据,只调用模块的 create_loc_conf,只用一个数组持有所有 http 模块在本 location 里的配置数据。

它的实现在 ngx_http_core_location() 函数里,代码基本相同,为节约篇幅不再列出。

location 层次配置数据的存储可以用图 6-6 来表示。

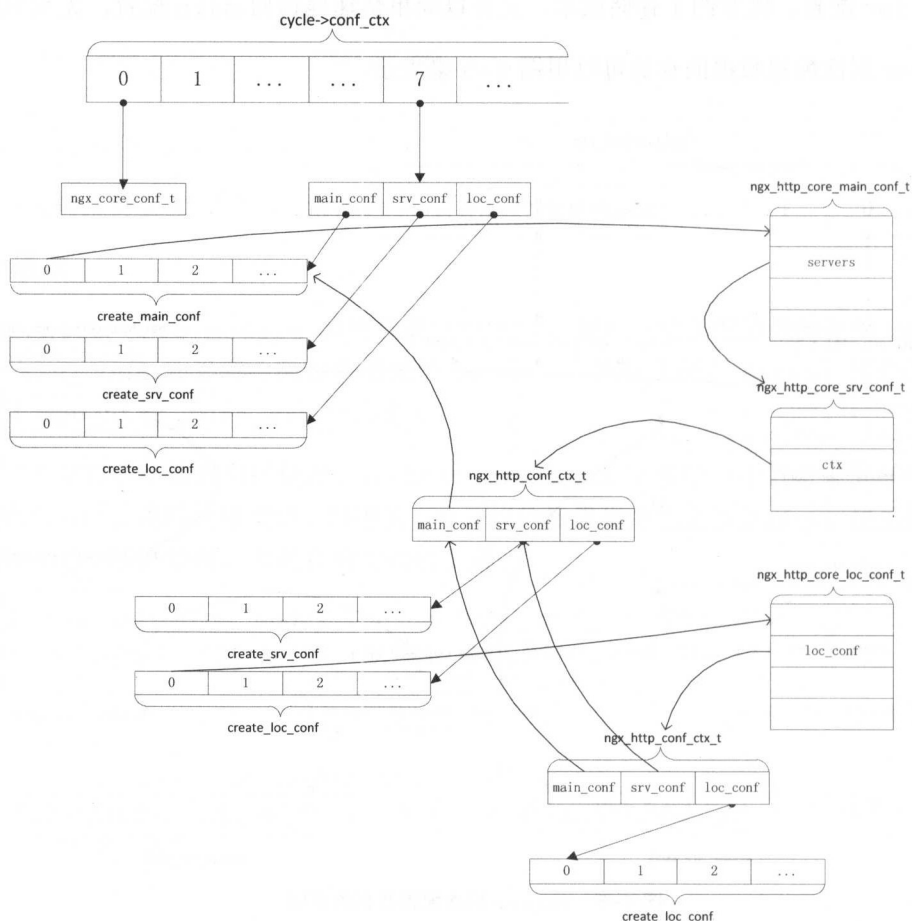


图 6-6 location 层次配置数据的存储

6.2.4 访问配置数据

通过 6.2.3 节的分析, 我们看到 Nginx 使用多个 `ngx_http_conf_ctx_t` 结构表示不同的配置层次, 每个结构代表一个块的配置, 所有 http 模块的配置数据都存储在结构里的 `main_conf/srv_conf/loc_conf` 数组里, 并且每个数组都以直接或者间接的方式存储了模块相关配置信息, 所以可以用 “`ctx.xxx_conf[module.ctx_index]`” 的形式访问任意 http 模块当前的正确配置数据, 完全不需要关心当前所在的配置块位置。

为了简化操作, Nginx 定义了三个宏, 用来在配置阶段直接获得 http 模块在当前 ctx 下的配置数据:

```
#define ngx_http_conf_get_module_main_conf(cf, module) \
    ((ngx_http_conf_ctx_t *) cf->ctx)->main_conf[module.ctx_index]

#define ngx_http_conf_get_module_srv_conf(cf, module) \
    ((ngx_http_conf_ctx_t *) cf->ctx)->srv_conf[module.ctx_index]

#define ngx_http_conf_get_module_loc_conf(cf, module) \
    ((ngx_http_conf_ctx_t *) cf->ctx)->loc_conf[module.ctx_index]
```

另外, 在处理 HTTP 请求时, Nginx 框架也会把模块配置数组存放在 `ngx_http_request_t` 对象里 (参见第 17 章):

```
struct ngx_http_request_s {
    void**      main_conf;      //http 层次的配置数组
    void**      srv_conf;       //server 层次的配置数组
    void**      loc_conf;       //location 层次的配置数组
    ...
};
```

所以可以用同样的方法获取模块的配置:

```
#define ngx_http_get_module_main_conf(r, module) \
    (r)->main_conf[module.ctx_index]

#define ngx_http_get_module_srv_conf(r, module) \
    (r)->srv_conf[module.ctx_index]

#define ngx_http_get_module_loc_conf(r, module) \
    (r)->loc_conf[module.ctx_index]
```

6.2.5 确定配置数据的位置

在 6.2.4 节, Nginx 调用 `create_xxx_conf` 为模块的配置数据分配好了存储空间 (三

个数组), 接下来的工作就是解析配置指令, 为模块找到正确的位置存储, 这需要由 `ngx_command_t` 的成员 `conf` 和 `offset` 来共同决定。

存储位置宏

在 `http/server/location` 三个层次里都使用的是 `ngx_http_conf_ctx_t` 结构, 里面有三个数组用于存放模块的配置数据。这三个数组都是完全相同的 `void*[]` 类型, 所以需要有一种方式来决定数据存放在哪一个数组里, 或者说是配置数据所在的作用域。

Nginx 在这里使用了结构偏移量, 也就是 `ngx_command_t` 结构里的 `conf` 成员, 让用户指定存储的位置 (作用域):^①

```
#define NGX_HTTP_MAIN_CONF_OFFSET  offsetof(ngx_http_conf_ctx_t, main_conf)
#define NGX_HTTP_SRV_CONF_OFFSET  offsetof(ngx_http_conf_ctx_t, srv_conf)
#define NGX_HTTP_LOC_CONF_OFFSET  offsetof(ngx_http_conf_ctx_t, loc_conf)
```

这三个宏利用 `offsetof` 获取了数组成员在 `ngx_http_conf_ctx_t` 结构内部的偏移量, Nginx 在解析指令时就可以使用偏移量得到要操作的数组:

```
//位于 core/nginx_conf_file.c 的 ngx_conf_handler()
confp = *(void **) ((char *) cf->ctx + cmd->conf); //偏移量获得数组

if (confp) { //检查数组是否有效
    conf = confp[cf->cycle->modules[i]->ctx_index]; //得到之前创建的配置数据
}
```

代码里的 `cf->ctx` 是当前解析指令所在层次的 `ngx_http_conf_ctx_t` 结构, 指令的配置数据存储在当前层次的三个数组里, 所以先用指令里的偏移量 `cmd->conf` 获得正确的数组指针, 再用 `ctx_index` 访问数组就可以得到。

存储偏移量

找到了配置数据结构的存储位置后, Nginx 还必须用偏移量字段 `offset` 才能访问到结构里的成员, 存放真正的数据。

偏移量字段是一个巧妙的设计, 可以让配置解析函数即使不知道配置数据的内部细节也能够直接定位变量的具体地址, 以这种方式来操作用户定义结构的“黑盒”, 实现去耦合。

偏移量字段不是解析所必需的, 如果我们自己编写解析函数, 结构定义已知, 那么这个字

^① 实际上, `conf` 成员是为 `stream/http` 模块专门设计的, 其他类型的模块因为没有复杂的层次关系, 所以不会用到这个成员。

段就可以设置为 0，在函数里直接操作数据成员。

6.2.6 配置解析函数

在 `ngx_command_t` 里配置解析函数 `set()` 的原型是：

```
char* (*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
```

它有三个参数：

- `cf` 是当前指令解析的环境数据，它的 `args` 成员存储了指令的所有参数；
- `cmd` 是当前指令的描述信息，也就是 `ngx_command_t` 数组里的元素；
- `conf` 是在当前层次的 `ngx_http_conf_ctx_t` 数组里存放的配置数据结构。

通常使用 `cf` 和 `conf` 这两个参数，再加上一些字符串处理代码，就可以完成对一条指令的解析工作。

注意它的返回值不是 `ngx_int_t`，而是 `char*`，如果出错可以直接返回错误信息的字符串，否则需要返回 `NGX_CONF_OK`，即 `NULL`。

预定义的解析函数

Nginx 预定义了 14 个配置解析函数，下面列出几个常用的函数。这些函数的名称均是“`ngx_conf_xxx_slot`”的形式，为了节约篇幅仅列出函数名中间的部分：

- `set_flag` : 处理值 `on|off`，转化为数字 `1|0`；
- `set_str` : 处理一个参数，转化为 `ngx_str_t`；
- `set_str_array` : 处理多个参数，转化为 `ngx_str_t` 数组；
- `set_keyval` : 处理多个参数，转化为 `ngx_keyval_t` 数组；
- `set_num` : 处理一个参数，转化为整数；
- `set_size` : 处理一个参数，转化为整数，可以使用单位 `K/M`；
- `set_msec` : 处理一个参数，转化为毫秒数，可以使用单位 `m/h/d/w/M/y` 等；
- `set_sec` : 类似 `set_msec`，但转化为秒数。

在使用这些预定义的配置解析函数时变量必须初始化为 `UNSET` 值，不能是简单的 0 或者空指针。

解析函数的实现

参考 Nginx 预定义的配置解析函数实现代码有助于我们开发自己的解析函数，这里以最

简单的 ngx_conf_set_flag_slot() 函数为例, 代码如下: ①

```
char *
ngx_conf_set_flag_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    char *p = conf;                                //指向配置数据结构体

    ngx_str_t      *value;                          //字符串指针
    ngx_flag_t      *fp;                            //存储变量的位置指针

    fp = (ngx_flag_t *) (p + cmd->offset);          //使用偏移量得到存储位置

    if (*fp != NGX_CONF_UNSET) {                    //检查 UNSET 值
        return "is duplicate";                      //返回错误信息
    }

    value = cf->args->elts;                          //获取参数字符串数组

    //大小写无关比较 on|off 字符串, 设置值为 1|0
    if (ngx_strcasecmp(value[1].data, (u_char *) "on") == 0) {
        *fp = 1;
    } else if (ngx_strcasecmp(value[1].data, (u_char *) "off") == 0) {
        *fp = 0;
    } else {
        ngx_conf_log_error(...);
        return NGX_CONF_ERROR;
    }

    return NGX_CONF_OK;
}
```

6.2.7 配置数据的合并

解析完所有的配置指令后, Nginx 的配置解析工作并没有结束, http/server/location 等多个层次里的配置数据数组还需要执行合并操作, Nginx 会逐层次地调用模块的 merge_srv_conf 和 merge_loc_conf 函数指针, 把高层次的值作为默认值赋给低层次, 最终实现配置的跨层次生效。

在 ngx_http_module_t 里这两个函数指针的定义是:

```
char *(*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);
```

① 这里省略了 post 字段的处理代码。


```
char *(*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
```

函数的第一个参数 `cf` 通常不会使用；参数 `prev` 表示上个层次的配置数据；`conf` 表示当前层次的配置数据，由于是 `void*` 所以需要先转型为配置数据结构才能做合并操作。

合并操作的通常做法是逐个检查当前结构里的值，如果未设置 (UNSET)，那么就默认使用高层次的对应值初始化。

配置数据的合并操作必须由我们自己实现，因为 Nginx 不知道我们的数据结构，也不知道我们的合并逻辑。不过 Nginx 预定义了 10 个配置合并宏，实现了通用的合并逻辑，实际上就是在 3.2.3 节介绍的条件赋值宏。

6.2.8 配置指令的类型

最后我们介绍 `ngx_command_t` 的成员 `type`，它是一个标志量，确定了指令所能出现的作用域、参数的数量和类型。

Nginx 使用一系列的宏来定义指令类型，这些宏实际上是二进制位，可以用逻辑或操作符 “|” 灵活组合来表达复杂的配置约束。

决定作用域的宏定义如下：^①

```
//定义在 http/ngx_http_config.h
#define NGX_HTTP_MAIN_CONF    0x02000000    //可以出现在 http 块里
#define NGX_HTTP_SRV_CONF    0x04000000    //可以出现在 server 块里
#define NGX_HTTP_LOC_CONF    0x08000000    //可以出现在 location 块里
#define NGX_HTTP_UPS_CONF    0x10000000    //可以出现在 upstream 块里
```

在 Nginx 的指令解析函数 `ngx_conf_handler()` 里会使用当前配置解析的环境数据 `ngx_conf_t::cmd_type` 来检查 `ngx_command_t::type`，如果不匹配则表明位置错误：

```
if (!(cmd->type & cf->cmd_type)) {           //检查指令的位置
    continue;                               //不匹配则忽略
}
```

决定参数数量的宏定义如下：

```
//定义在 core/ngx_conf_file.h
#define NGX_CONF_NOARGS    0x00000001    //指令没有参数
#define NGX_CONF_TAKE1    0x00000002    //指令有 1 个参数
#define NGX_CONF_TAKE2    0x00000004    //指令有 2 个参数
```

^① 这里省略了 `event`、`stream` 等模块使用的宏。

```

#define NGX_CONF_TAKE3      0x00000008      //指令有 3 个参数
#define NGX_CONF_TAKE4      0x00000010      //指令有 4 个参数
#define NGX_CONF_TAKE5      0x00000020      //指令有 5 个参数
#define NGX_CONF_TAKE6      0x00000040      //指令有 6 个参数
#define NGX_CONF_TAKE7      0x00000080      //指令有 7 个参数

#define NGX_CONF_MAX_ARGS    8                //指令最多有 8 个参数

//指令可以有 1~N 个参数，是上面宏的组合
#define NGX_CONF_TAKE12      (NGX_CONF_TAKE1|NGX_CONF_TAKE2)
#define NGX_CONF_TAKE13      (NGX_CONF_TAKE1|NGX_CONF_TAKE3)

```

决定参数类型的宏定义如下：

```

//定义在 core/nginx_conf_file.h
#define NGX_CONF_FLAG        0x00000200      //参数取值为 on|off，只能有一个
#define NGX_CONF_ANY         0x00000400      //不限制参数的数量
#define NGX_CONF_1MORE       0x00000800      //参数数量必须超过 1 个
#define NGX_CONF_2MORE       0x00001000      //参数数量必须超过 2 个

```

在 5.1.2 节里 `ndg_test_module` 模块定义的类型是：

```

NGX_HTTP_LOC_CONF|NGX_CONF_FLAG      //配置为 HTTP_LOC+FLAG

```

那么它的含义就是允许指令出现在 `location` 域，并且参数取值为 `on|off`。

如果想让指令在其他域里也可以出现，那么就使用“|”来连接更多的宏，例如：

```

//允许指令出现在 http 块的任意位置，但 upstream 块除外
NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG

```

6.3 源码分析

本节简要研究 `ngx_core_module` 和 `ngx_errlog_module` 这两个模块的源码，但并不解析其具体工作逻辑，仅仅是展现模块的定义和配置的解析部分。结合 6.1 节和 6.2 节可以更好地理解 Nginx 的模块架构和配置解析，同时也可以了解到 `core` 模块并不神秘，甚至比大多数 `http` 模块还要简单。

6.3.1 ngx_core_module

`ngx_core_module` 是 Nginx 最基本的模块，在所有的模块里它的编号是 0，是 `ngx_modules` 数组里的第一个模块，定义了 `daemon`、`master_process` 和 `worker_processes` 等核心指令，至关重要。

ngx_core_module 定义在文件 nginx.c 里，而配置数据结构 ngx_core_conf_t 则定义在 ngx_cycle.h 里。

配置结构定义

ngx_core_conf_t 结构保存了 Nginx 运行所需的基本参数，与配置指令一一对应，摘要如下：

```
typedef struct {
    ngx_flag_t    daemon;           //守护进程标志位
    ngx_flag_t    master;           //master 进程标志位
    ngx_int_t     worker_processes; //worker 进程的数量
    ...                          //其他成员变量
} ngx_core_conf_t;
```

指令数组

ngx_core_commands 定义了 ngx_core_module 使用的配置指令，同时指定了指令的解析函数：

```
static ngx_command_t  ngx_core_commands[] = {

    { ngx_string("daemon"),           //守护进程指令
      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_FLAG,
      ngx_conf_set_flag_slot,         //简单地解析 on|off 的函数
      0,
      offsetof(ngx_core_conf_t, daemon),
      NULL },

    { ngx_string("master_process"),   //master 进程指令
      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_FLAG,
      ngx_conf_set_flag_slot,         //简单地解析 on|off 的函数
      0,
      offsetof(ngx_core_conf_t, master),
      NULL },

    ...                               //其他指令

    ngx_null_command                  //指令数组结束
};
```

这些指令的 type 字段都是 NGX_MAIN_CONF|NGX_DIRECT_CONF，只能出现在配置文件最外层的 main 域（不是 http main 域）。

创建和初始化配置结构

依据 core 模块的 ctx 要求，ngx_core_module 需要两个函数，用来创建和初始化配

置数据结构 ngx_core_conf_t。

ngx_core_module_create_conf() 函数的代码与我们的 ndg_test_module 没有什么区别，同样是分配内存，并置值为 UNSET：

```
static void *
ngx_core_module_create_conf(ngx_cycle_t *cycle)
{
    ngx_core_conf_t *ccf;                //配置结构数据指针

    ccf = ngx_palloc(                     //创建配置结构数据
        cycle->pool, sizeof(ngx_core_conf_t));

    ccf->daemon = NGX_CONF_UNSET;        //置为 UNSET
    ...                                  //其他 UNSET
    return ccf;
}
```

因为 ngx_core_module 的指令在配置文件的最外层，没有合并的需求，所以在配置文件解析结束后要用 ngx_core_module_init_conf() 函数来给出配置的默认值，相当于 ngx_http_module_t 的 init_main_conf 函数指针的作用：

```
static char *
ngx_core_module_init_conf(ngx_cycle_t *cycle, void *conf)
{
    ngx_core_conf_t *ccf = conf;        //配置结构数据指针

    ngx_conf_init_value(ccf->daemon, 1); //默认启用守护模式
    ngx_conf_init_value(ccf->master, 1); //默认启用 master 进程
    ...                                  //其他操作
    return NGX_CONF_OK;
}
```

模块定义

ngx_core_module 使用 ngx_core_module_t 结构集成创建和初始化函数：

```
static ngx_core_module_t ngx_core_module_ctx = {
    ngx_string("core"),                //模块名字
    ngx_core_module_create_conf,        //创建配置结构
    ngx_core_module_init_conf           //初始化配置结构
};

ngx_module_t ngx_core_module = {      //模块定义
    NGX_MODULE_V1,
    &ngx_core_module_ctx,              //module context
    ngx_core_commands,                 //module directives
}
```

```

    NGX_CORE_MODULE, //module type
    ...              //省略其他变量
};

```

业务逻辑

ngx_core_module 自身没有业务逻辑操作,它只是提供了 Nginx 运行所需的基本参数, Nginx 框架在运行时直接获取并使用,例如:

```

ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
if (!ngx_inherited && ccf->daemon) { ... }

```

这里的 ngx_get_conf 是一个很简单的宏,用来从 conf_ctx 数组里获取配置结构:

```

#define ngx_get_conf(conf_ctx, module) conf_ctx[module.index]

```

6.3.2 ngx_errlog_module

ngx_errlog_module 是 ngx_modules 数组里的第二个模块,它紧挨着 ngx_core_module,定义在 ngx_log.c。

与 ngx_core_module 不同的是它不需要额外的配置信息,所以没有创建和初始化函数。

指令数组

ngx_errlog_module 只定义了一个指令 error_log,所以指令数组非常简单:

```

static ngx_command_t ngx_errlog_commands[] = {
    {ngx_string("error_log"), //运行日志指令
     NGX_MAIN_CONF|NGX_CONF_1MORE,
     ngx_error_log, //指令解析函数
     0,
     0,
     NULL},
    ngx_null_command
};

```

模块定义

ngx_errlog_module 的模块定义也很简单:

```

static ngx_core_module_t ngx_errlog_module_ctx = {
    ngx_string("errlog"), //模块名字
    NULL, //不需要配置结构
    NULL
};

```

```

ngx_module_t ngx_errlog_module = {                                //模块定义
    NGX_MODULE_V1,
    &ngx_errlog_module_ctx,                                       //module context
    ngx_errlog_commands,                                         //module directives
    NGX_CORE_MODULE,                                             //module type
    ...                                                            //省略其他变量
};

```

指令解析函数

`ngx_error_log()` 是 `ngx_errlog_module` 的主要功能函数，它解析指令，设置 `ngx_cycle_t` 里的 `log` 变量：

```

char *
ngx_log_set_log(ngx_conf_t *cf, ngx_log_t **head)
{
    value = cf->args->elts;                                       //获取配置字符串
    ...                                                            //省略内部操作
    return NGX_CONF_OK;
}

```

6.4 C++封装

在熟悉了 Nginx 的模块架构和配置解析机制之后，本节将使用 C++ 对这些 C 接口进行封装，简化 Nginx 的模块开发工作。

但因为 Nginx 的配置使用的是 `ngx_command_t`、`ngx_http_module_t` 等结构，必须手工设置很多的细节，没有明显的内聚性，配置解析的封装类只能一定程度上减轻这些工作量。

6.4.1 NgxModuleConfig

在 6.1 节可以看到，`ngx_module_t` 结构里最重要的成员就是 `ctx_index`，在编写 http 模块代码的很多时候都需要用它来获取配置信息，所以我们实现一个 `NgxModuleConfig` 类，它相当于增强的 `ngx_http_conf_get_module_xxx_conf` 系列宏。

类定义

由于模块的配置有 `http/server/location` 三个层次，而每个层次又可能使用不同类型的配置结构体，为了避免客户代码里 `void*` 类型转换的麻烦，`NgxModuleConfig` 直接把配置结构体作为模板参数，在编译期固化下来。

`NgxModuleConfig` 的类定义如下：

```
template<typename T3 = void, typename T2 = void, typename T1 = void>
class NgxModuleConfig final
{
public:
    NgxModuleConfig(ngx_uint_t idx): m_idx(idx)           //获取模块的序号
    {}
    ~NgxModuleConfig() = default;
private:
    ngx_uint_t m_idx = 0;                                //保存模块的序号
    ...                                                    //其他成员函数，见后
};
```

NgxModuleConfig 并没有从 NgxWrapper 继承，这是因为它的目标很明确，只用来获取模块的配置，所以用一个 m_idx 成员来保存模块序号足矣，不需要其他额外信息。

NgxModuleConfig 使用了三个模板参数，分别表示 loc_conf/srv_conf/main_conf 里存储的配置信息。读者需要注意模板里的参数顺序，第一个模板 T3 表示的是 loc_conf，而最后一个模板 T1 表示的是 main_conf。因为大多数情况下我们都使用 location 层次的配置，只传递一个模板参数就足够了，可以简化代码的编写。

基本操作

NgxModuleConfig 的基本操作是获取 Nginx 配置数组里的数据结构：

```
public:
    ngx_uint_t index() const                                //获取模块序号
    {
        return m_idx;
    }

    template<typename T, typename U>
    T* get(U conf) const                                    //获取配置数据结构
    {
        return reinterpret_cast<T*>(&conf[index()]);
    }
```

get() 函数抽象了 ngx_http_conf_get_module_xxx_conf 系列宏的操作，以模板函数的形式给出了通用获取配置信息的方法，模板参数 U 可以是一个任意数组。

获取 ngx_conf_t 里的配置信息

ngx_conf_t 里的成员 ctx 类型是 void*，为了获取配置信息需要先转换为 ngx_http_conf_ctx_t 类型：

```
private:
```

```
ngx_http_conf_ctx_t* ctx(ngx_conf_t* cf) const
{
    return reinterpret_cast<ngx_http_conf_ctx_t*>(cf->ctx);
}
```

然后我们就可以调用 `get()` 函数, 使用 `NgxModuleConfig` 的模板参数类型直接获取三个数组里存储的模块配置信息:

```
public:
    T1* main(ngx_conf_t* cf) const           //获取 http 层次的配置信息
    {
        return get<T1>(ctx(cf)->main_conf); //使用模板参数 T1
    }

    T2* srv(ngx_conf_t* cf) const           //获取 sever 层次的配置信息
    {
        return get<T2>(ctx(cf)->srv_conf);  //使用模板参数 T2
    }

    T3* loc(ngx_conf_t* cf) const           //获取 location 层次的配置信息
    {
        return get<T3>(ctx(cf)->loc_conf);  //使用模板参数 T3
    }
```

获取 `ngx_http_request_t` 里的配置信息

同样的, 我们可以获取 `ngx_http_request_t` 里的配置信息, 因为 `ngx_http_request_t` 直接给出了三个数组指针, 所以代码更简单:

```
public:
    T1* main(ngx_http_request_t* r) const //获取 http 层次的配置信息
    {
        return get<T1>(r->main_conf);    //使用模板参数 T1
    }

    T2* srv(ngx_http_request_t* r) const //获取 sever 层次的配置信息
    {
        return get<T2>(r->srv_conf);      //使用模板参数 T2
    }

    T3* loc(ngx_http_request_t* r) const //获取 location 层次的配置信息
    {
        return get<T3>(r->loc_conf);      //使用模板参数 T3
    }
```


模板元编程

使用 C++ 的高级技术模板元编程, `NgxModuleConfig` 能够做得更好: 如果 `T` 是 `void`, 那么函数返回 `void*` 指针; 如果 `T` 是一个结构体, 那么函数返回 `T&`, 这样可以直接用点号 (“.”) 操作配置数据结构里的成员, 编写代码会更加方便。

首先我们需要修改 `get()` 函数, 当 `T` 是 `void` 时返回 `void*`, 否则返回 `T&`:

```
template<typename T, typename U>
typename std::enable_if<std::is_void<T>::value, void*>::type
get(U conf) const
{
    return conf[index()]; //直接返回 void*
}

template<typename T, typename U>
typename std::enable_if<std::is_object<T>::value, T&>::type
get(U conf) const
{
    return *reinterpret_cast<T*>(conf[index()]); //注意*的使用, 返回 T&
}
```

`get()` 函数使用了一个元函数 `enable_if`, 可以在编译期根据 `T` 的类型来判断是否允许模板函数实例化。当 `T` 是一个实体对象类型时, `get()` 会使用第二种形式, 对指针使用 `*` 操作符返回引用, 否则调用第一种形式返回指针。^①

因为 C++11 标准不支持函数类型的自动推导 (即函数的返回值使用 `auto`), 所以我们还要实现一个元函数 `eval_type`, 它计算函数应该返回的类型:

```
template<typename T> //计算 T 应该返回的类型
struct eval_type : //使用了元函数转发
    std::conditional< //标准条件元函数, 相当于 if
        std::is_void<T>::value, //检查类型是否是 void
        void*, //返回 void*
        typename std::add_lvalue_reference<T>::type> //返回 T&
    {};

//定义宏简化元函数的调用
#define META_TYPE(t) typename eval_type<t>::type
```

① 我们也可以使用 Boost 程序库的 `enable_if`、`disable_if` 等元函数, 代码会略简洁一些。但为了风格一致这里都使用了标准库里的元函数, 更多的模板元编程知识请参考推荐书目 [4]。

`eval_type` 的实现涉及比较深的模板元编程知识, 本书暂不做过多的介绍, 只需要知道 `eval_type` 可以在编译期计算类型, 根据模板参数 `T` 返回不同的类型就可以了。

最后我们需要把 `main()/srv()/loc()` 等函数的返回值类型改用 `META_TYPE` 宏, 例如:

```
META_TYPE(T1) main(ngx_http_request_t* r) const    //自动计算正确的返回值类型
{
    return get<T1>(r->main_conf);                    //返回值类型由 T1 确定
}

META_TYPE(T2) srv(ngx_http_request_t* r) const    //自动计算正确的返回值类型
{
    return get<T2>(r->srv_conf);                      //返回值类型由 T2 确定
}
```

用法

使用 `NgxModuleConfig` 的 `main()/srv()/loc()` 函数可以很轻松地获取模块的配置信息, 无须再做转型操作, 而且因为这三个函数分别对 `ngx_conf_t*` 和 `ngx_http_request_t*` 做了重载, 所以也不必再像使用 Nginx 自带宏那样去费心地选择, 绝对不会出现笔误的低级错误。

在调用 `main()/srv()/loc()` 函数时需要使用 `auto&` 的形式来获取配置结构的引用, 这是因为函数返回的是 `T&`, 而 `auto` 的推导原则是值类型, 直接使用 `auto` 会导致不必要的拷贝成本。^①

假设我们已经有了一个 `NgxModuleConfig` 对象, 那么获取配置信息的代码就是:

```
NgxModuleConfig<...> conf(...);                    //一个 NgxModuleConfig 对象
auto& loc = conf.loc(r);                            //获取 location 配置, 无须转型
auto& srv = conf.srv(r);                            //获取 server 配置, 无须转型
auto x = loc.xxx;                                    //获取 location 配置里的成员
```

当然, `NgxModuleConfig` 的模板参数的书写和对象的构造不是太容易, 它依赖于具体的 Nginx 模块, 这个工作可以由接下来的 `NgxModule` 类完成。

6.4.2 NgxModule

`NgxModule` 类封装了 `ngx_module_t` 结构, 它主要的功能是生产模块对应的 `NgxModuleConfig` 对象。

^① 在 C++14 里可以使用含义更明确的 `decltype(auto)`。

类定义

NgxModule 的实现比较简单，相当于一个工厂类，根据模块的 `ctx_index` 产生合适的 `NgxModuleConfig` 对象：

```
template<typename T3 = void, typename T2 = void, typename T1 = void>
class NgxModule //注意不是 final
{
public:
    NgxModule(ngx_module_t& m): //引用模块对象
        m_conf(m.ctx_index) //使用序号 ctx_index
    {}

    ~NgxModule() = default;
private:
    typedef NgxModuleConfig<T3, T2, T1> config_type; //简化类型定义

    config_type m_conf; //NgxModuleConfig 对象
public:
    const config_type& conf() const //产生 NgxModuleConfig
    {
        return m_conf;
    }
};
```

NgxModule 类没有使用准关键字 `final`，这是为了后续扩展的考虑，某些特定的模块可以复用 `NgxModule` 的代码，增加模块自己的独有功能。^①

对于 5.1 节的模块 `ndg_test_module`，`NgxModule` 可以这样使用：

```
typedef NgxModule<NdgTestConf> mod_type; //定义具体的模块类型
mod_type mod(ndg_test_module); //代理 Nginx 模块
auto& cf = mod.conf().loc(r); //获取 location 配置
```

在第 7 章，我们还会扩充 `NgxModule` 的功能，通过它获取到模块的其他信息。

模块定义宏

`NgxModule` 虽然简化了模块的配置获取，但用起来还有一点儿不方便，必须每次用 `ngx_module_t` 变量构造后才能使用。最好的方法是提供一个全局的可任意访问的单件，把 `NgxModule<T>` 和具体的 `ngx_module_t` 模块绑定在一起。

^① 第 7 章的 `NgxHttpCoreModule` 就使用了这种方法，封装了模块 `ngx_http_core_module`。

为了达成这个目的，我们定义一个工具宏 `NGX_MOD_INSTANCE`，它为每个 Nginx 模块产生一个单件类：

```
#define NGX_MOD_INSTANCE(T, mod, ...) \           //单件生成宏
struct T { \                                     //单件类名
    typedef ngx_module_t __VA_ARGS__ mod_type; \ //ngx_module 类型定义
    static mod_type& instance() \               //单件访问函数
    { \
        extern ngx_module_t mod; \             //外部的模块变量
        static mod_type m(mod); \              //单件对象
        return m; \                            //返回单件引用
    } \
    static const config_type& conf() \         //简化单件的调用
    { return instance().conf(); } \            //直接返回配置对象
};
```

宏 `NGX_MOD_INSTANCE` 定义了一个单件类，代码也很简单，先 typedef 了 `ngx_module` 类型，然后声明外部的模块变量，构造对象并返回。

它使用了 C99 里的可变参数宏的语言特性，可以接受三个或者更多的参数，第一个是单件类的名字，第二个是模块的变量名，其他则是模块的配置结构类型，用法是：

```
NGX_MOD_INSTANCE(XModule, xxx_module, XConf) \ //定义单件
auto& cf = XModule::conf().loc(r); \           //获取 location 配置
```

6.4.3 ngxTake

类 `ngxTake` 封装了 Nginx 里参数数量的设置，用两个整数来表示参数的数量，减少宏的出现。

类定义

`ngxTake` 使用 C++ 的默认参数特性，在构造函数里完成工作，使用转型操作符：

```
class ngxTake final
{
public:
    ngxTake(ngx_uint_t conf, int m, int n = -1): \ //构造函数
        m_type(conf | take(m, n)) \              //基本参数+数量
    {}
    ~ngxTake() = default;
public:
    operator ngx_uint_t () const \               //转型操作符
    {
        return m_type; \                         //返回类型标志变量
    }
};
```

```
private:
    ngx_uint_t m_type = NGX_HTTP_LOC_CONF;    //类型标志变量
    ...                                       //其他成员函数
};
```

计算参数数量

NgxTake 的 take() 函数把参数上下限 m/n 转换为 Nginx 的宏:

```
private:
    static ngx_uint_t take(int m, int n)
    {
        static
        ngx_uint_t takes[] = {               //Nginx 的 TAKE 宏
            NGX_CONF_NOARGS,
            NGX_CONF_TAKE1,
            ...
            NGX_CONF_TAKE7,
        };

        if(n < 0 || n < m)                   //参数有效性检查
        {
            return takes[m];
        }

        if(n >= NGX_CONF_MAX_ARGS)          //参数数量太大
        {
            return m == 1 ?
                NGX_CONF_1MORE :             //多于 1 个
                NGX_CONF_2MORE;              //多于 2 个
        }

        ngx_uint_t tmp = 0;
        for(int i = m; i <= n; ++i)          //逻辑或多个宏
        {
            tmp |= takes[i];
        }

        return tmp;
    }
};
```

用法

NgxTake 仅仅封装了参数数量的设置, 并不包含其他的 NGX_CONF_FLAG、NGX_HTTP_

LOC_CONF 等宏, 这些宏需要在构造函数的 conf 参数里设置, 例如:

```
NgxTake(NGX_HTTP_LOC_CONF|NGX_CONF_FLAG, 1) //使用一个参数, flag 类型
```

6.4.4 NGX_MODULE_NULL

ngx_http_module_t 和 ngx_module_t 结构里有很多字段不是必须的, 虽然 Nginx 提供了 NGX_MODULE_V1 等宏, 但在初始化时还有一些函数指针需要使用 nullptr 填充, 显得比较麻烦。

仿造 Nginx 的方式, 我们也可以定义宏专门用于简化空函数指针的设置。简单的做法是定义 NULL2、NULL3 这样的宏, 每个宏里有 N 个空指针, 但使用 Boost 提供的预处理元编程库可以得到更优雅的解法:

```
#include <boost/preprocessor/repetition/enum.hpp> //预处理元编程头文件

#define NGX_NULL_HELPER(z, n, t) t //辅助宏
#define NGX_MODULE_NULL(n) \ //实际使用的宏
    BOOST_PP_ENUM(n, NGX_NULL_HELPER, nullptr) //预处理元编程
```

这里我们使用了 boost.preprocessor 库的 BOOST_PP_ENUM 元函数, 它可以迭代 n 次, 调用辅助宏 NGX_NULL_HELPER 产生多个逗号分隔的字符串。例如, NGX_MODULE_NULL(3) 在经过预处理后会变成:

```
nullptr, nullptr, nullptr //产生三个 nullptr
```

6.5 C++开发模块

本节将使用前面的 C++ 类重新实现 5.1 节的 ndg_test_module, 编写一个真正现代 C++ 风格的 Nginx 模块, 读者可以对比看看两者有哪些差异。

6.5.1 模块的基本组成

使用 C 语言编写 Nginx 模块, 源码的组成通常比较简单, 因为模块都是独立的, 所以只需在一个 *.c 里实现指令解析、业务逻辑和模块集成等所有功能代码, 再加上几个头文件, 源文件的组织形式如图 6-7 所示。^①

^① 这里说的是逻辑较为简单的 Nginx 模块, 如果模块需要实现的业务很复杂, 当然也会分成多个 .c 和 .h 文件。

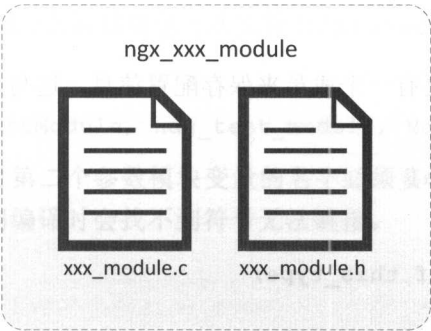


图 6-7 C 语言源文件的组织形式

由于 C++ 提供了面向对象特性，能够以类封装不同的功能模块，所以最好不要采用 C 语言的简单组织形式，而是把不同的功能代码分别放在不同的源文件里。

本书采用下面的方式：

- *XConf.hpp* 实现配置数据及相关操作；
- *XHandler.hpp* 实现业务逻辑；
- *XInit.hpp* 实现指令配置和模块集成；
- *ModX.cpp* 定义 `ngx_module_t` 全局模块变量。

C++ 源文件的组织形式如图 6-8 所示。

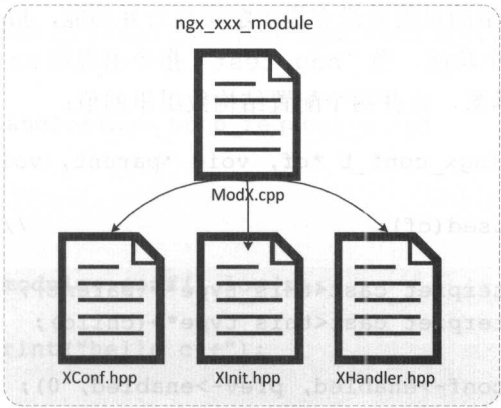


图 6-8 C++ 源文件的组织形式

6.5.2 配置信息类

我们使用面向对象的思想重新实现类 `NdgTestConf`，除基本的配置信息外还要增加创建和合并等操作。

类定义

NdgTestConf 仍然需要有一个成员来保存配置信息，这与 5.1 节相同：

```
// NdgTestConf.hpp
class NdgTestConf final //禁止被继承
{
public:
    typedef NdgTestConf this_type; //自身类型定义
public:
    ngx_flag_t enabled = ngx_nil; //标志变量，构造时初始化
    ... //其他成员函数，见后
};
```

创建函数

静态成员函数 create() 实现了 ngx_http_module_t 结构所需的创建配置信息功能：

```
public:
    static void* create(ngx_conf_t* cf) //创建配置数据结构
    {
        return NgxPool(cf).alloc<this_type>(); //直接内存池分配内存
    }
```

合并函数

5.1 节的模块出于简单的目的并没有提供配置项合并功能，指令只能在 location 里出现。现在我们为它增加这个功能。当“ndg_test”指令出现在 http 块或者 server 块中时 Nginx 会调用 merge() 函数，合并两个配置结构数组里的值：

```
static char* merge(ngx_conf_t *cf, void *parent, void *child)
{
    boost::ignore_unused(cf); //不使用 cf 变量，消除警告

    auto prev = reinterpret_cast<this_type*>(parent); //上层的配置结构
    auto conf = reinterpret_cast<this_type*>(child); //本层的配置结构

    NgxValue::merge(conf->enabled, prev->enabled, 0); //合并变量，默认值 0

    return NGX_CONF_OK;
}
```

模块单件定义

由于已经有了配置信息类 NdgTestConf，所以可以使用 6.4.2 节的宏 NGX_MOD_

INSTANCE 来定义模块的单件类,之后只要包含头文件 `NdgTestConf.hpp` 就可以随时获取模块的配置信息:

```
NGX_MOD_INSTANCE(NdgTestModule, ndg_test_module, NdgTestConf)
```

注意:在定义模块单件时第二个参数模块变量的名字必须要与最终在 `cpp` 里的模块变量名一致(即 6.5.5 节),否则编译时会找不到符号无法链接。

6.5.3 业务逻辑类

`NdgTestHandler` 实现了模块的处理函数和框架注册:

```
// NdgTestHandler.hpp
class NdgTestHandler final
{
public:
    typedef NdgTestHandler this_type;           //自身类型定义
    typedef NdgTestModule this_module;         //模块类型定义
public:
    ...                                         //其他成员函数, 见后
};
```

处理函数

`NdgTestHandler` 的处理函数是静态成员函数 `handler()`, 代码与 5.1 节基本相同, 但增加了捕获可能发生异常的 `try-catch` 操作和 `NgxClock` 计时:

```
public:
    static ngx_int_t handler(ngx_http_request_t *r)    //业务逻辑函数
    try
    {
        NgxClock clock;                               //计时器计算时间

        auto& cf = this_module::conf().loc(r);         //获取配置

        NgxLogError(r).print("hello c++");            //记录运行日志

        if (!cf.enabled)                               //检查模块是否禁用
        {
            cout << "hello disabled" << endl;         //输出字符串
            return NGX_DECLINED;                       //执行成功但未处理
        }

        cout << "hello nginx" << endl;                //输出字符串
```

```

std::cout << clock.elapsed() << "s"<< std::endl;    //输出时间

return NGX_DECLINED;                                //执行成功但未处理
}
catch(const NgxException& e)                         //捕获可能发生的异常
{
    return e.code();                                //返回错误码
}

```

注册函数

首先我们要为 `ngx_http_core_module` 定义代理类，因为它的三个配置数据结构都不同，所以宏有五个参数：^①

```

NGX_MOD_INSTANCE(NgxHttpCoreModule, ngx_http_core_module,
                  ngx_http_core_loc_conf_t,
                  ngx_http_core_srv_conf_t, ngx_http_core_main_conf_t)

```

静态成员函数 `init()` 向 Nginx 框架注册 `handler()`，使用 `NgxHttpCoreModule` 类可以轻松地获取配置数据：

```

public:
static ngx_int_t init(ngx_conf_t* cf)
{
    auto& cmcf = NgxHttpCoreModule::conf().main(cf);

    typedef NgxArray<ngx_http_handler_pt> handler_array_t;

    handler_array_t arr(cmcf.phases[NGX_HTTP_REWRITE_PHASE].handlers);
    arr.push(&this_type::handler);

    return NGX_OK;                                //执行成功
}

```

注意：在注册 `handler` 时静态成员函数指针的使用方式必须要加取地址操作符“&”，这是与自由函数明显不同的地方。

6.5.4 模块集成类

类 `NdgTestInit` 整合 `NdgTestConf` 和 `NdgTestHandler`，实现 Nginx 模块集成。

① 在之后的第7章我们会进一步封装 `ngx_http_core_module`，让框架注册工作更加简单。

类定义

NdgTestInit 的定义如下:

```
// NdgTestInit.hpp
class NdgTestInit final
{
public:
    typedef NdgTestConf      conf_type;           //简化类型定义
    typedef NdgTestHandler   handler_type;
    typedef NdgTestInit      this_type;
public:
    ...                                           //其他成员函数，见后
};
```

配置指令解析

静态成员函数 `cmds()` 创建 `ngx_command_t` 数组，它使用了 6.4.3 节的 `NgxTake`，减少了编写的代码量:

```
public:
    static ngx_command_t* cmds()
    {
        static ngx_command_t n[] =                //配置指令数组，静态变量
        {
            {
                ngx_string("ndg_test"),           //指令的名字
                NgxTake(                           //指令的作用域和类型
                    NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG, 1),
                ngx_conf_set_flag_slot,           //解析函数指针
                NGX_HTTP_LOC_CONF_OFFSET,         //数据的存储位置
                offsetof(conf_type, enabled),      //数据的具体存储变量
                nullptr
            },
            ngx_null_command                       //空对象，结束数组
        };

        return n;                                //返回数组地址
    }
```

与 5.1 节的代码相比，函数 `cmds()` 变化并不是很大，这是由 Nginx 模块架构所决定的。

函数指针表

静态成员函数 `ctx()` 使用 `NdgTestConf` 和 `NdgTestHandler` 的成员函数填充 `ngx_`

http_module_t 结构, 指示 Nginx 框架在配置解析时需要调用的函数:

```
public:
    static ngx_http_module_t* ctx()
    {
        static ngx_http_module_t c =
        {
            NGX_MODULE_NULL(1),           //解析配置文件前被调用, 不使用
            &handler_type::init,          //解析配置文件后被调用
            NGX_MODULE_NULL(4),           //不使用的 4 个函数指针
            &conf_type::create,           //创建 location 域的配置结构
            &conf_type::merge,            //合并 location 域的配置结构
        };

        return &c;                        //注意返回的是指针
    }
```

这里使用了 6.4.4 节的宏 NGX_MODULE_NULL, 不需要再逐个写出 nullptr。

模块定义

最后, 我们用 ctx() 和 cmds() 定义 ngx_module_t 对象:

```
public:
    static const ngx_module_t& module()
    {
        static ngx_module_t m =           //模块定义, 静态变量
        {
            NGX_MODULE_V1,                 //标准的填充宏
            ctx(),                          //配置功能函数
            cmds(),                         //配置指令数组
            NGX_HTTP_MODULE,               //http 模块必须的 tag
            NGX_MODULE_NULL(7),            //不使用的 7 个函数指针
            NGX_MODULE_V1_PADDING          //标准的填充宏
        };

        return m;                          //返回对象的常量引用
    }
```

6.5.5 实现源文件

ndg_test_module 的所有功能都已经实现, 但它们仅存在于头文件里, 要想编译进 Nginx 还必须要有个 cpp 文件。

ModNdgTest.cpp 的代码非常简单, 只有一行, 定义 Nginx 编译所需的 ngx_module_t 变量,

这也恰当地反映了 Nginx 模块的本质——`ngx_module_t` 就是模块：

```
#include "NdgTestInit.hpp"           //包含头文件
auto ndg_test_module = NdgTestInit::module(); //定义 ngx_module_t 变量
```

由于模块名称和源文件名没有变动，所以 `config` 脚本无须修改，可以直接使用“`--add-module`”或“`--add-dynamic-module`”参数把模块集成进 Nginx，再使用 `curl` 测试，步骤与 5.1.6 节相同。

6.5.6 增加更多功能

现在我们已经有了第一个可以运行的 Nginx 模块 `ndg_test_module`，虽然它目前还不能做什么有实际意义的工作，基本是一个空的模块，但它毕竟能够嵌入 Nginx 框架里运行，所以完全可以进一步扩展它的功能，把它作为一个单元测试的容器，编写测试代码，学习验证 Nginx 的各种数据结构 and 接口函数。

例如，可以在 `NdgTestHandler` 里编写一个 `timer()` 函数测试 `NgxDatetime`：

```
static void timer()
{
    cout << NgxDatetime::today();           //输出当天日期
    cout << NgxDatetime::http();           //输出当天日期的 http 格式

    auto t = NgxDatetime::since();          //获取当前时间戳
    auto s = NgxDatetime::http(t);          //转换 http 字符串格式
    assert(t == NgxDatetime::http(s));      //验证字符串转时间戳
    ...                                     //更多的测试语句
}
```

然后在 `handler()` 函数里执行：

```
static ngx_int_t handler(ngx_http_request_t *r) //业务逻辑函数
{
    ...                                     //其他功能
    timer();                                //执行测试函数
}
```

6.6 总结

本章探究了 Nginx 里最核心、最重要的模块体系，包括模块架构和配置解析，这两者的联系非常密切，构成了整个 Nginx 宫殿，每一个模块就是宫殿里的一个房间。

模块架构

可以把 Nginx 的配置文件想象成是一种领域特定语言 (domain special language), 而 Nginx 模块架构则是这种语言的解析器, 编写 Nginx 模块就是为这种语言增加词汇和语义动作, 不断地丰富词汇就能够让 Nginx 掌握更多的本领, 变得更加强大。

`ngx_module_t` 结构体描述了 Nginx 的模块, 为了能够实现灵活健壮的模块架构, 它保留了一些字段供将来使用, 开发中常用的是 `ctx_index`、`type`、`commands` 和 `ctx` 四个字段。`ctx_index` 记录了模块的访问序号, `type` 标记了模块的类型, `commands` 定义了模块支持的指令集合, 而 `ctx` 则是模块架构可扩展性的根本, 实现了类似 C++ 的继承特性, 可以派生出无数种类的模块。对于 http 模块来说, `ctx` 是一个函数指针的集合, 这些函数指针会在 Nginx 解析配置的某个时间点被调用, 可以执行添加变量、初始化、创建配置结构体等功能。

1.9.11 之后的 Nginx 引入了动态模块特性, 它很大程度上改变了 Nginx 模块开发的模式, 内部的 C 代码、外部的编译脚本也都发生了显著的变化。为了减少对众多第三方模块的影响, Nginx 利用了填充宏 `NGX_MODULE_V1`, 修改了它的旧定义, 使原有的第三方模块几乎不需要变动就能与新版本 Nginx 兼容 (但必须要重新编译)。

动态模块的好处是实现了 Nginx 主程序与功能模块的分离, 使两者可以各自独立维护, 但分离也带来了二进制兼容的问题。Nginx 在 `ngx_module_t` 结构体里使用新字段 `signature` 来生成与操作系统、编译环境相关的模块“签名”, 在启动阶段对比 `signature`, 执行兼容性检查。

早期所有的模块都存储在全局数组 `ngx_modules` 里, 在 1.9.11 之后它仅用来保存静态模块, 启动完成之后就被“废弃”, 目前所有的模块 (动态和静态) 都存储在 `cycle->modules` 这个数组里, 由 `ngx_preinit_modules()` 等一系列模块专用函数管理维护。

配置解析

读取并解析配置文件在很多系统里都是一件无足称道的工作, 很多程序仅把它视为简单的纯文本或者略微复杂的 xml/ini, 只有在需要的时候才去获取配置数据。而 Nginx 的高明之处就是把配置文件与模块架构紧密地结合在一起, 启动阶段模块负责解析配置文件, 逐层次地在内存里建立完全对应的内存存储模型, 这样运行的时候就无须频繁地访问文件, 可以直接在内存里高效便捷地获取模块的配置。

Nginx 为配置文件设计了精巧的存储结构, 最顶层是 `ngx_cycle_t::conf_ctx`, 它只存储 core 模块的配置。为了支持 HTTP 服务的虚拟主机和虚拟目录, Nginx 使用了 `ngx_http_conf_ctx_t` 结构, 它内部有三个数组, 用来存储 `http/server/location` 三个层次的配置。

每一个 `http/server/location` 块都会持有一个 `ngx_http_conf_ctx_t` 结构，这些结构之间使用指针互相连接，表述块的包含关系，形成了一个具有复杂结构的树，树的根则是 `http` 块的 `ngx_http_conf_ctx_t` 结构——因为配置文件里只能有一个 `http` 块定义。

解析 Nginx 配置文件主要的工作就是构建 `http` 配置树。在创建树前执行 `pre-configuration` 做准备，然后调用 `create_xxx_conf` 填充节点数组，遇到 `server` 和 `location` 指令再创建新的节点。叶子节点则是具体的指令，由具体的模块负责解释。当解析完毕时，配置树也创建完成，这时就要遍历树执行 `init` 和 `merge` 操作，逐个设置配置数据的值，避免出现 `UNSET`，最后执行 `postconfiguration` 完成收尾工作。

`ngx_command_t` 结构定义了解析配置指令的方式，成员 `conf` 确定了模块配置数据结构存储的数组，`offset` 确定了结构里变量的具体位置。这样函数指针 `set` 就可以使用 `conf` 找到数组里的结构体对象，再用 `offset` 找到结构体里的成员，然后从 `cf->args` 里获取字符串，执行配置动作。成员 `type` 是用来确定指令的作用域、类型和参数的标志量，可以使用逻辑或操作组合很多特性，Nginx 在解析流程里会参考 `type` 来检查指令是否正确。

我们还简略研究了 `ngx_core_module` 和 `ngx_errlog_module` 这两个核心模块，通过阅读官方代码，可以更好地理解 Nginx 的模块架构和配置解析机制。

C++封装

本章的后半部分是模块的 C++实现。

`NgxModuleConfig` 和 `NgxModule` 封装了模块最常用的获取配置信息操作，它们利用 C++的模板特性，在编译期确定了 `main/srv/loc` 三个层次的配置类型，从而简化了客户代码，不需要再执行转型操作，直接使用关键字 `auto` 或 `decltype` 就可以得到正确的对象。

因为每个 Nginx 模块都是唯一的，符合单件的概念，所以我们实现了宏 `NGX_MODULE_INSTANCE`，它可以非常简单地为每个 Nginx 模块产生一个单件类，提供唯一的全局访问点。

在配置解析方面，本章提供了 `NgxTake` 和 `NGX_MODULE_NULL` 两个工具，它们给 Nginx 的原始接口施加了一层简单的封装，虽然并没有做太多的工作，但确实可以减少编写实际模块的代码量。

最后我们重新实现了第 5 章的 `ndg_test_module` 模块，把模块分解为 `NdgTestConf`、`NdgTestHandler` 和 `NdgTestInit` 三个类，分别对应配置解析、业务逻辑和模块集成三个功能，再在 `ModNdgTest.cpp` 里定义 `ngx_module_t` 变量，完成了纯 C++风格的 Nginx 模块。

学习建议

本章的内容较多，需要仔细耐心地阅读才能理解掌握，读者可以结合第13章、第17章了解 Nginx 进程机制、启动过程和 HTTP 机制等加深体会。

第 7 章

Nginx HTTP 框架综述

在第 5、6 章里我们学习了 Nginx 模块开发流程，仔细研究了 Nginx 的模块架构和配置解析机制，具备了编写 Nginx 模块的基本能力。本章将在此基础上进一步探讨 Nginx HTTP 处理框架，也就是最常用的 http 模块的工作原理，剖析 Nginx 的 HTTP 处理流程和相关数据结构。

7.1 框架简介

Nginx 的 HTTP 框架是由 core 模块 `ngx_http_module` 和 http 模块 `ngx_http_core_module` 共同定义的。

`ngx_http_module` 定义了指令 `http`，保存和管理各个层次里所有 http 模块的配置数据（我们已经在第 6 章了解了它的工作原理）；而 `ngx_http_core_module` 则是 http 模块里的“核心”模块，它定义了 `listen`、`server`、`location` 等 http 核心指令，搭建了 Nginx 的 HTTP 处理框架。

7.1.1 模块分类

http 模块处理现今应用得最广泛的 HTTP 协议，也是目前 Nginx 里数量最多的模块，按照功能分类可以分为四种：

- `handler` : 直接处理客户端的请求，产生响应数据，是最常用的模块；
- `filter` : 对 `handler` 模块产生的数据做各种加工过滤处理；
- `upstream` : 实现反向代理功能，转发请求到上游服务器，从后端获取响应数据再发回给客户端；

- `load-balance` : 不直接处理数据, 而是实现负载均衡算法, 从 `upstream` 块的配置里选择一个合适的上游服务器。

这四种模块的工作机制各不相同, 本章重点讨论最复杂也是最常用的 `handler` 模块和 `filter` 模块, 后两种模块留给第 9 章。

7.1.2 处理流程

本节简要介绍 Nginx 的 HTTP 工作流程, 这是理解 Nginx HTTP 框架的基础。

通用的处理流程

Web 服务器的 HTTP 请求处理流程可以粗略地叙述为如下的顺序步骤:

- 1) 监听端口, 接受客户端的连接;
- 2) 读取请求头, 包括请求行 (`request line`) 和请求头 (`headers`);
- 3) 读取或者丢弃请求体 (`body`);
- 4) 生成并发送响应头;
- 5) 生成并发送响应体。

上面只是最简单的 HTTP 处理流程, Nginx 使用了操作系统提供的事件驱动模型(`epoll`、`kqueue` 等) 来实现异步并发, 并且还增加了很多其他功能, 如权限检查、重定向、缓存、记录访问日志等, 使 HTTP 处理流程变得十分复杂。

Nginx 的处理流程

如果我们忽略 Nginx 的异步机制, 把注意力集中在处理逻辑上, 那么可以看到 Nginx 的主处理流程与基本的 HTTP 处理流程区别并不大, 它的工作流程如下:^①

- 1) 监听端口, 设置回调为 `ngx_http_init_connection()`。
- 2) 接受客户端的连接, 调用 `ngx_http_wait_request_handler()`。
- 3) 调用 `ngx_http_create_request()` 创建请求对象。
- 4) 接收数据, 调用 `ngx_http_process_request_line()` 解析请求行。
- 5) 请求行接收完毕, 继续接收数据, 调用 `ngx_http_process_request_headers()` 解析请求头。
- 6) 请求头接收完毕, 调用 `ngx_http_process_request()` 设置异步读写事件。注意:

① 当前版本的 Nginx 中 HTTP 处理流程与早期版本略微不同, 少量函数做了改变, 例如函数 `ngx_http_init_request()` 已经不复存在。

为了提高运行效率，Nginx 不会主动处理请求体。

- 7) 调用 ngx_http_handler() 开始真正处理请求。
- 8) 调用 ngx_http_core_run_phases() 按阶段处理请求。这是 HTTP 框架的核心部分，大部分 http 模块都在这里运行，最终产生响应数据。
- 9) 调用 ngx_http_send_header() 发送响应头，从函数指针 ngx_http_top_header_filter 开始，通过 header filter 模块链过滤处理，最终发送处理过的响应头。
- 10) 调用 ngx_http_output_filter() 发送响应体，从函数指针 ngx_http_top_body_filter 开始，通过 body filter 模块链过滤处理，最终发送处理过的响应体。
- 11) 处理完毕，记录访问日志。

Nginx HTTP 处理流程图如图 7-1 所示。

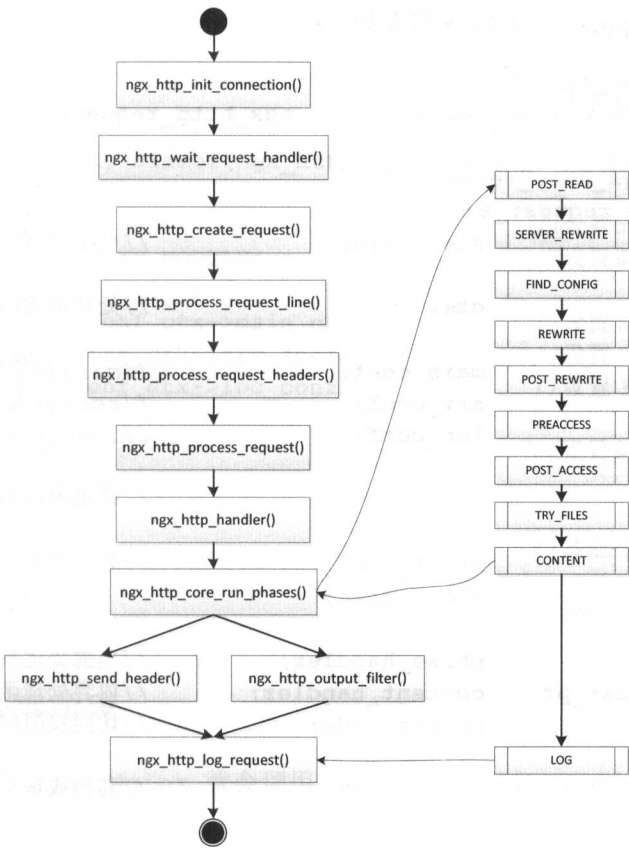


图 7-1 Nginx HTTP 处理流程图

在这些步骤中比较重要的是第3步和第8步：

`ngx_http_create_request()` 创建请求对象 `ngx_http_request_t`，初始化里面的内存池、起始时间、配置结构数组等成员，应用于整个 HTTP 处理生命周期。

`ngx_http_core_run_phases()` 分阶段地组织所有的 http 模块（准确地说是 handler 模块），调用模块的处理函数，执行虚拟主机查找、重定向、权限检查、内容产生等各种操作，是 HTTP 框架运行机制中最根本的部分，掌握了它也就掌握了 Nginx 的 HTTP 框架。

7.1.3 请求结构体

在第6章我们初步接触了结构体 `ngx_http_request_t`，它表示一个 HTTP 请求，是 Nginx 处理 HTTP 请求的核心数据结构，包含处理过程中需要的各种数据和信息，比较复杂。

`ngx_http_request_t` 的定义摘要如下：

```
// 定义在 http/nginx_http.h
typedef struct ngx_http_request_s      ngx_http_request_t;

// 定义在 http/nginx_http_request.h
struct ngx_http_request_s {
    uint32_t          signature;           // "HTTP" 的 ASCII 码

    void**            ctx;                 // 处理请求的环境数据

    void**            main_conf;           // main 层次的配置信息数组
    void**            srv_conf;           // server 层次的配置信息数组
    void**            loc_conf;           // location 层次的配置信息数组

    ngx_pool_t*       pool;                // 请求使用的内存池

    time_t            start_sec;           // 处理请求的开始时间
    ngx_msec_t        start_msec;

    ngx_int_t         phase_handler;       // 当前处理阶段的索引号
    ngx_http_handler_pt content_handler;   // 内容处理函数指针
    ngx_uint_t        access_code;        // 访问权限码

    ...                                     // 其他成员
};
```

`ngx_http_request_t` 贯穿了 Nginx 处理 HTTP 请求的整个流程，可以说在 Nginx 中

处理 HTTP 请求就是操作 `ngx_http_request_t` 数据结构。

由于 `ngx_http_request_t` 里的成员很多，因此本章只介绍与 Nginx HTTP 框架有关的部分，其他成员如请求行、请求头、请求体等将在第 8 章介绍。

`ngx_http_request_t` 里有几个成员我们已经在第 6 章见过，例如 `main_conf/srv_conf/loc_conf`，它们分别保存了 `http` 模块在配置文件中对应的配置信息，这样在处理 HTTP 请求时就不必访问配置文件，可以直接获取。

`ngx_http_create_request()` 创建 `ngx_http_request_t` 对象，初始化它的各个成员，代码摘要如下：

```
//定义在 http/ngx_http_request.c
ngx_http_request_t *
ngx_http_create_request(ngx_connection_t *c)           //创建请求对象
{
    ngx_http_request_t      *r;                       //结构体指针

    ...                                                //其他操作代码

    r = ngx_palloc(pool, sizeof(ngx_http_request_t)); //创建结构体对象

    r->pool = pool;                                     //设置请求的内存池
    r->signature = NGX_HTTP_MODULE;                    //设置请求的“签名”

    r->main_conf = hc->conf_ctx->main_conf;             //设置 main 配置指针
    r->srv_conf = hc->conf_ctx->srv_conf;               //设置 srv 配置指针
    r->loc_conf = hc->conf_ctx->loc_conf;               //设置 loc 配置指针

    r->start_sec = tp->sec;                             //设置起始时间
    r->start_msec = tp->msec;

    ...                                                //其他操作代码

    return r;                                           //返回创建好的对象
}
```

7.1.4 请求的处理阶段

在接收完请求头数据后，Nginx 就会调用 `ngx_http_core_run_phases()` 驱动各个 `http` 模块处理请求。

为了能够更细粒度地控制 HTTP 处理流程，Nginx 在这里划分出了 11 个精确的阶段（或

者说是 hook 点), 模块可以挂载到不同的阶段, 相互协作以流水线的方式处理请求, 从而达到高度的灵活性。

可以把阶段理解为一种高级形式的调用接口, 它里面可以存储数个回调函数, Nginx 会逐个调用里面的函数, 完成请求的处理。

Nginx 使用枚举定义了 11 个请求处理阶段, 它们是: ①

```
//定义在 http/nginx_http_core_module.h
typedef enum {
    NGX_HTTP_POST_READ_PHASE = 0,           //读取 HTTP 头后, 开始读取内容

    NGX_HTTP_SERVER_REWRITE_PHASE,         //server 重写 URL

    NGX_HTTP_FIND_CONFIG_PHASE,             //查找匹配的 location, 不可介入
    NGX_HTTP_REWRITE_PHASE,                 //location 重写 URL
    NGX_HTTP_POST_REWRITE_PHASE,            //重写后, 不可介入

    NGX_HTTP_PREACCESS_PHASE,               //检查访问权限前

    NGX_HTTP_ACCESS_PHASE,                  //检查访问权限
    NGX_HTTP_POST_ACCESS_PHASE,             //检查访问权限后, 不可介入

    NGX_HTTP_TRY_FILES_PHASE,               //访问静态文件, 不可介入

    NGX_HTTP_CONTENT_PHASE,                //处理请求, 产生内容

    NGX_HTTP_LOG_PHASE                      //记录日志
} ngx_http_phases;                         //枚举类型定义
```

ngx_http_phases 定义的这 11 个阶段含义都很明确, 完整地描述了 HTTP 请求的处理流程。所有的 handler 模块都必须工作在这 11 个阶段中的一个或几个, 所以 handler 模块又称为 phase handler 模块。

但有四个阶段只能由 HTTP 框架使用, 用户不能注册开发模块在这些阶段处理请求 (即使注册了也会被框架忽略), 它们是:

- NGX_HTTP_FIND_CONFIG_PHASE
- NGX_HTTP_POST_REWRITE_PHASE

① 实际上还有请求体和响应体数据的过滤阶段, 但过滤是在 ngx_http_core_run_phases() 外通过职责链的方式实现的, 参见 7.3 节。

- NGX_HTTP_POST_ACCESS_PHASE
- NGX_HTTP_TRY_FILES_PHASE

所以我们开发的模块只能介入剩下的 7 个阶段。

REWRITE、ACCESS 和 CONTENT 是开发 Nginx 模块比较常用的阶段，可以改写 URI 和权限控制，或者产生我们自己的响应内容。

7.1.5 请求的环境数据

ngx_http_request_t 里的 ctx 是 HTTP 请求处理流程中一个非常重要的成员，它的类型是 void**，即 void* 的数组，为每一个 http 模块都提供了专属的存储空间，模块可以在里面存储任意数据，整个请求的生命周期里都可以随时访问，就像是个全局变量。

在函数 ngx_http_create_request() 里创建 ctx 的代码如下：

```
r->ctx = ngx_palloc(r->pool,           //创建环境数据数组
    sizeof(void *) * ngx_http_max_module); //长度是所有 http 模块的数量
```

ctx 与 loc_conf 等都是 void** 类型，但两者存储的数据有根本性的区别：loc_conf 等存储的是静态的配置数据，与配置的层次相关，与请求无关；ctx 存储的是动态的运行期数据，与配置的层次无关，与请求紧密相关。

由于 Nginx 的异步和多阶段处理机制的原因，很多时候 http 模块不可能在一次调用中就完成请求的处理，所以可以用 ctx 来暂时存储处理的中间结果，当 Nginx 再次调用模块时再从 ctx 中取出上次的数据继续运行——这实际上是备忘录模式 (memento) 的具体应用。^①

模块不能简单地使用全局变量来暂存数据。这是因为数据是请求相关的，每个请求的状态都不一样，而 Nginx 会并发处理大量的请求，单纯的全局变量不能应对这样复杂的情况。Nginx 把 ctx 和请求绑定在一起，解决了这个问题。

我们可以使用模块的 ctx_index 直接访问 ctx 数组获取模块的环境数据，不过 Nginx 也提供了两个宏来操作 ctx 环境数据，用法与获取配置结构信息类似：

```
//定义在 http/ngx_http.h
#define ngx_http_get_module_ctx(r, module) (r)->ctx[module.ctx_index]
#define ngx_http_set_ctx(r, c, module)      r->ctx[module.ctx_index] = c;
```

如果模块需要在处理过程中保存一些中间结果，通常的做法是定义一个 struct，里面包

① 也可以理解成“断点恢复”。

含所有要存储的信息，然后使用内存池创建对象，调用 `ngx_http_set_ctx()` 加入 `ctx` 数组，之后就可以随时使用 `ngx_http_get_module_ctx()` 来获取并操作数据。

创建环境数据对象时一定要使用请求对象 `ngx_http_request_t` 的内存池成员 `r->pool`，这样当请求结束时分配的内存才会被及时回收。

7.2 处理引擎

Nginx 的模块 `ngx_http_core_module` 管理着所有的 handler 模块，它定义了请求的处理阶段，把所有的 handler 模块组织成一条流水线，使用一个“引擎”来驱动模块处理 HTTP 请求，产生响应内容，本节将详细阐述这一机制。

7.2.1 函数原型

Nginx 定义了函数原型 `ngx_http_handler_pt`，任何 handler 模块想要处理 HTTP 请求都必须实现这个函数，它的形式是：

```
//定义在 http/nginx_http_request.h
typedef ngx_int_t (*ngx_http_handler_pt) (ngx_http_request_t *r);
```

`ngx_http_handler_pt` 不仅可以返回 `NGX_OK`、`NGX_DECLINED`、`NGX_AGAIN` 等 Nginx 错误码，也可以直接返回 200、302、404 等标准 HTTP 状态码，Nginx 会在函数 `ngx_http_finalize_request()` 中做出合适的处理。

7.2.2 处理函数的存储方式

Nginx 使用 `ngx_http_phase_t` 结构存储每个阶段可用的处理函数 (handler)，它实际上是动态数组 `ngx_array_t`，元素类型是 `ngx_http_handler_pt`：

```
//定义在 http/nginx_http_core_module.h
typedef struct {
    ngx_array_t          handlers;           //处理函数数组
} ngx_http_phase_t;
```

`ngx_http_core_module` 的 main 配置结构又定义了数组 `phases`，用来存储处理阶段中所有可用的 handler：

```
//定义在 http/nginx_http_core_module.h
typedef struct {
    ...
    ngx_http_phase_t phases[NGX_HTTP_LOG_PHASE + 1]; //处理阶段数组
```



```
} ngx_http_core_main_conf_t;
```

通过这种方式，Nginx 把所有 http 模块的 handler 组织成了一个表，纵向是处理阶段，横向是各个模块的处理函数，形成了一个二维职责链，如图 7-2 所示。

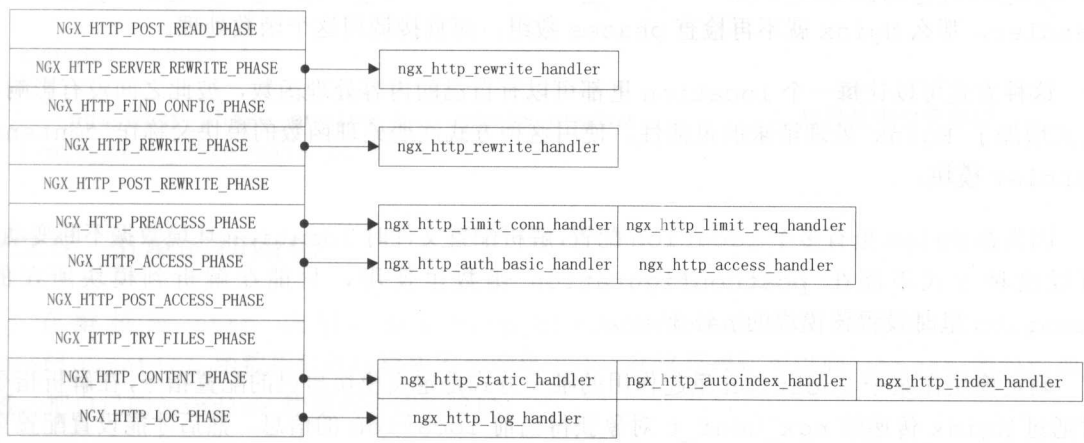


图 7-2 handler 二维职责链

在配置文件解析完毕之后的 `postconfiguration` 函数里，模块可以向 `ngx_http_core_main_conf_t::phases` 数组添加元素，实现模块的 handler 注册，就像第 6 章我们所做的。

处理 HTTP 请求时 Nginx 框架将会分阶段按顺序调用这些函数，让每个模块都有执行的机会——不过实际的代码远不是双重循环那么简单。

7.2.3 内容处理函数

`phases` 数组是模块注册 handler 最通用的方式，可以介入任意阶段，但产生内容的 `NGX_HTTP_CONTENT_PHASE` 阶段则有其特殊性。

由于很多模块都会工作在 `CONTENT` 阶段，如果都使用添加到 `phases` 数组的方式很可能会导致数组元素过多，运行效率低，也很容易导致模块冲突。所以 Nginx 单独为内容处理函数提供另外一处存储位置。

`ngx_http_core_module` 的 `location` 配置数据结构 `ngx_http_core_loc_conf_t` 里有一个成员 `handler`，它直接确定了该 `location` 的处理函数：

```
//定义在 http/ngx_http_core_module.h
struct ngx_http_core_loc_conf_s {
    ngx_http_handler_pt handler;
    //location 配置结构
    //该 location 的处理函数
};
```

```
... //其他成员
};
```

Nginx 查找 location 时会检查 `loc_conf` 的 `handler` 成员, 如果不为空那么就会设置 `ngx_http_request_t` 的 `content_handler`。在 `CONTENT` 阶段如果有 `content_handler`, 那么 Nginx 就不再检查 `phases` 数组, 而直接调用这个函数处理。

这种方式可以让每一个 location 里都可以有自己的内容处理函数, 彼此之间没有影响, 大大增加了 Nginx 处理请求的灵活性。使用这种方式注册处理函数的模块又称作 `content handler` 模块。

因为在 Nginx 里有多个 location 配置, 解析配置文件时 location 环境数据不断变动, 所以这种方式不能在 `postconfiguration` 函数里使用, 只能在解析到模块所在的 location 里时设置该模块的 `handler`。

确定模块在某个 location 里起作用的唯一方式是定义模块自己的配置指令, 在解析指令时通过 Nginx 传递的 `ngx_conf_t` 对象获得当前 location 的信息, 然后才能设置配置里的 `handler`, 这也就意味着我们至少要编写一个自定义的配置解析函数。

7.2.4 引擎的数据结构

使用二维数组 `phases` 可以调用所有 `handler` 模块, 但它的组织形式不够灵活, 效率也不高。实际上, Nginx 不会直接调用 `handler`, 而是为每个阶段实现一个特定的 `checker` 函数, 在 `checker` 里调用 `handler`, 并根据返回值实现阶段的灵活跳转。

`checker` 的相关定义如下:

```
//阶段处理数据结构
typedef struct ngx_http_phase_handler_s  ngx_http_phase_handler_t;

//checker 的函数原型
typedef ngx_int_t (*ngx_http_phase_handler_pt)(
    ngx_http_request_t *r, ngx_http_phase_handler_t *ph);

struct ngx_http_phase_handler_s {
    ngx_http_phase_handler_pt  checker;           //检查器函数
    ngx_http_handler_pt        handler;          //模块的处理函数
    ngx_uint_t                 next;             //下一处理阶段的序号
};

ngx_http_core_main_conf_t 里使用 ngx_http_phase_engine_t 存储了所有
handler 模块的 checker, 定义了 HTTP 框架的处理引擎:
```

```

typedef struct {
    ngx_http_phase_handler_t*  handlers;           //包含所有模块的 handler
    ngx_uint_t                  server_rewrite_index; //server 重写快速跳转索引
    ngx_uint_t                  location_rewrite_index; //location 重写快速跳转索引
} ngx_http_phase_engine_t;                        //处理引擎定义

typedef struct {
    ...
    ngx_http_phase_engine_t    phase_engine;       //阶段处理引擎数组
} ngx_http_core_main_conf_t;

```

7.2.5 引擎的初始化

在解析完 http 块后，ngx_http_block() 函数会调用所有模块的 post-configuration 函数指针执行模块的初始化工作，在这里模块可以向 phases 数组添加元素，把自己的 handler 注册到合适的阶段。

随后 Nginx 执行函数 ngx_http_init_phase_handlers()，从 phases 数组生成处理引擎 phase_engine。

ngx_http_init_phase_handlers() 会遍历 phases 数组，计算 handler 模块的数量，统计所有已经注册的 handler 数量并分配内存^①，再按阶段分类，把每个 handler 与对应阶段的 checker 组合起来，填入引擎数组，部分关键代码如下：

```

ph = ngx_pcalloc(...);           //引擎数组分配内存
n = 0;                             //计数器初始化

for (i = 0; i < NGX_HTTP_LOG_PHASE; i++) { //按阶段遍历，不含 log 阶段
    h = cmcf->phases[i].handlers.elts;      //本阶段的 handler 数组

    switch (i) {                             //根据阶段选择 checker
        ...                                  //省略其他阶段的代码

        case NGX_HTTP_SERVER_REWRITE_PHASE: //rewrite 阶段
            checker = ngx_http_core_rewrite_phase; //该阶段使用的 checker
            break;

        case NGX_HTTP_CONTENT_PHASE:         //content 阶段
            checker = ngx_http_core_content_phase; //该阶段使用的 checker

```

① 这里虽然也计算了 FIND_CONFIG 等特殊阶段的数量，但在之后并不会真正加入引擎数组。

```
break;

default:
    checker = ngx_http_core_generic_phase;    //默认的 checker
}

//反向遍历 phases 数组，向引擎数组添加元素
for (j = cmcf->phases[i].handlers.nelts - 1; j >=0; j--) {
    ph->checker = checker;    //设置 checker
    ph->handler = h[j];    //设置 handler
    ph->next = n;    //设置阶段跳转序号
    ph++;
}

//phases 数组遍历结束
```

在 ngx_http_init_phase_handlers() 执行之后，phase_engine 的 handlers 数组把各个模块的 handler 组成了一个线性的数组，原本的二维数组 phases 转变成了一维数组，可以高效地访问。而每个元素 (ngx_http_phase_handler_t 结构) 里的 next 形成了静态链表，指示了阶段跳转的数组序号，也就是下一个阶段的起始位置，通过它可以跳过本阶段的其他 handler，直接进入下个处理阶段。

初始化后的引擎数组可以用图 7-3 来表示。

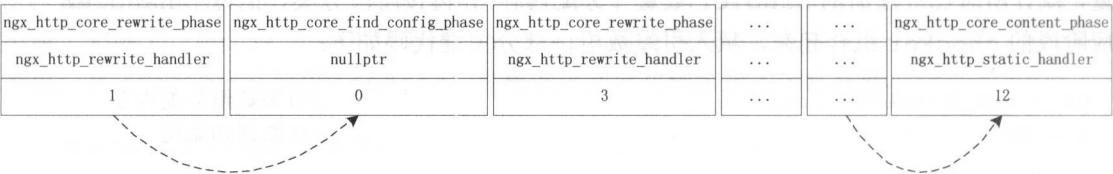


图 7-3 初始化后的引擎数组

还需要注意的一点是，phase_engine 数组并不含有 NGX_HTTP_LOG_PHASE 阶段的模块，这是因为记录日志阶段比较特殊，它并不真正地处理请求，是请求处理的收尾工作，不需要跳转到其他阶段。

7.2.6 引擎的运行机制

结构体 ngx_http_request_t 里的成员 phase_handler 标记了在当前处理过程中所在阶段的 handler 序号，它被初始化为 0，通常是 SERVER_REWRITE 阶段里的第一个模块

ngx_http_rewrite_module。^①

接收完请求头后，Nginx 调用函数 ngx_http_core_run_phases() 执行引擎，代码摘要如下：

```
ph = cmcf->phase_engine.handlers;           //获取引擎数组首地址

while (ph[r->phase_handler].checker) {       //遍历引擎数组

    rc = ph[r->phase_handler].checker(        //执行 checker,间接调用 handler
        r, &ph[r->phase_handler]);
    if (rc == NGX_OK) {                       //检查返回值
        return;                              //如果执行成功则暂时退出引擎
    }
}
```

ngx_http_core_run_phases() 的代码比较简单，它遍历引擎数组，检查数组里的 checker，如果有则执行之。在 checker 函数里会调用 handler，并决定是否要在引擎数组里依次执行或者跳转。如果 checker 返回任何非 NGX_OK 的错误码，那么就表示请求还未处理完毕，需要在引擎数组里查找模块处理；否则 Nginx 会暂时退出处理引擎，等待下次发生网络 ready 事件时继续运行。

ngx_http_core_run_phases() 的流程图如图 7-4 所示。

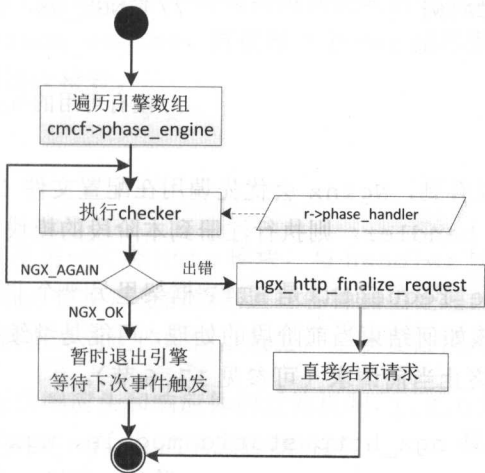


图 7-4 ngx_http_core_run_phases() 的流程图

① 理论上引擎数组里的第一个模块应该是 NGX_HTTP_POST_READ_PHASE 阶段，但实际上很少有模块注册在这个阶段。

不同阶段的 checker 流程大同小异（详细的讲解见第 17 章），这里我们以 CONTENT 阶段使用的 `ngx_http_core_content_phase()` 为例，看一下它的主要处理逻辑（省略了部分代码，详细的流程图参见 17.3.5 节）：

```

ngx_int_t
ngx_http_core_content_phase(ngx_http_request_t *r,
    ngx_http_phase_handler_t *ph)
{
    if (r->content_handler) {                //检查是否有 location 内容处理函数
        ngx_http_finalize_request(           //直接调用内容处理函数
            r, r->content_handler(r));
        return NGX_OK;                       //执行完毕，退出引擎
    }

    rc = ph->handler(r);                      //没有 location 特定的处理函数，执行此模块的 handler

    if (rc != NGX_DECLINED) {                //其他错误码
        ngx_http_finalize_request(r, rc);    //以合适的方式结束请求
        return NGX_OK;                      //执行完毕，退出引擎
    }

    ph++;                                    //模块返回 NGX_DECLINED，需要继续执行本阶段的下一个 handler
    if (ph->checker) {                       //下一个模块有 checker
        r->phase_handler++;                  //调整请求里的阶段索引号
        return NGX_AGAIN;                   //非 NGX_OK，要求引擎继续执行
    }

    return NGX_OK;                          //没有可用的 handler，退出引擎
}

```

从这段代码里我们可以看到，Nginx 会优先调用在配置文件 location 里设置的处理函数，如果没有 location handler，则执行注册到本阶段的模块 handler。

`ngx_http_finalize_request()` 是 HTTP 框架里另一个非常重要的函数。它根据错误码或 HTTP 状态码决定应该如何结束当前阶段的处理，可能是继续执行处理引擎，也可能是暂时退出引擎，还有可能是终止当前请求（可参见 17.6 节）。

通常 Nginx 会默认编译 `ngx_http_static_module`、`ngx_http_index_module` 等模块，它们会注册到 `NGX_HTTP_CONTENT_PHASE` 阶段，所以一个 location 如果没有设置特殊的内容处理函数就会由这几个模块处理，读取磁盘上的文件。

从这里我们还可以明确 `NGX_OK`、`NGX_DECLINED`、`NGX_AGAIN` 等错误码在内容生成阶段的真实含义：^①

- `NGX_OK` : 模块正确运行, 请求处理成功, 不需要再有其他动作, 引擎结束;
- `NGX_DECLINED` : 模块正确运行, 请求未处理, 需要下一个模块继续处理;
- `NGX_AGAIN` : 模块正确运行, 请求部分处理, 需要引擎继续运行。

7.2.7 日志阶段的处理

日志阶段的模块不在 `ngx_http_core_run_phases()` 里调用, 而是在请求处理完毕时才会执行, 函数是 `ngx_http_log_request()`:

```
static void
ngx_http_log_request(ngx_http_request_t *r)
{
    log_handler = cmcf->phases[NGX_HTTP_LOG_PHASE].handlers.elts;
    n = cmcf->phases[NGX_HTTP_LOG_PHASE].handlers.nelts;

    for (i = 0; i < n; i++) {
        log_handler[i](r);
    }
}
```

因为在日志阶段请求已经处理完毕, 不需要再做其他数据处理工作, 所以日志模块的调用很简单, 不使用引擎数组 `phase_engine`, 直接在一个 `for` 循环里执行 `phases` 数组里所有的 `log handler`, 也不做错误检查。

7.3 过滤引擎

`filter` 模块是 Nginx 里另一大类 `http` 模块, 与 `handler` 模块不同的是它并不直接产生响应数据, 而是对 `handler` 模块产生的数据进行各种加工处理, 例如增加响应头、对响应体编码、`gzip` 压缩等, 可以更灵活地控制输出数据, 是职责链模式的实际应用。

早期版本的 Nginx 只提供响应头和响应体的过滤机制, 1.8.0 版后的 Nginx 增加了对请求体的过滤处理, 本节主要阐述 Nginx 的响应过滤机制。

^① 不同的阶段的 `checker` 函数逻辑不完全相同, 所以 `NGX_OK`、`NGX_DECLINED`、`NGX_AGAIN` 等错误代码的含义也不一定相同, 可参考第 17 章进一步了解。

7.3.1 函数原型

Nginx 把过滤处理细分为响应头处理和响应体处理两种，定义了两个函数类型：

```
//定义在 http/nginx_http_core_module.h
typedef ngx_int_t (*ngx_http_output_header_filter_pt) //过滤响应头
                  (ngx_http_request_t *r);
typedef ngx_int_t (*ngx_http_output_body_filter_pt)  //过滤响应体
                  (ngx_http_request_t *r, ngx_chain_t *chain);
```

从这两个函数的名字可以看出，它们是输出过滤器(output)，只能处理请求的响应内容。

函数 ngx_http_output_header_filter_pt 处理响应头，头信息已经包含在了 ngx_http_request_t 结构里，所以不需要其他参数。

函数 ngx_http_output_body_filter_pt 处理响应体，需要外部使用 chain 参数传递给它待输出的数据，这里的数据就是第4章介绍的 ngx_buf_t 和 ngx_chain_t 对象。

filter 模块必须实现这两个函数以达到过滤处理的目的，但如果只处理响应头，那么就无须实现 ngx_http_output_body_filter_pt。

7.3.2 过滤函数链表

Nginx 定义了过滤函数链表的头节点：

```
//定义在 http/nginx_http.c
ngx_http_output_header_filter_pt    ngx_http_top_header_filter;
ngx_http_output_body_filter_pt      ngx_http_top_body_filter;
```

这两个头节点是全局可见的，所以 Nginx 以一种隐晦却巧妙的方式把过滤函数组成了一个顺序链表：

- 1) 模块实现自己的过滤器函数 this_filter;
- 2) 模块定义一个变量 next_filter，作为指向下一个节点的指针；
- 3) 模块在 postconfiguration 阶段用 next_filter 存储 top_filter；
- 4) 模块设置 top_filter 为 this_filter。

写成代码就是：

```
next_filter = top_filter;           //存储 top_filter
top_filter = this_filter;          //设置头节点
```

可以看出，这是一种典型的单链表添加头节点操作。

通过这种方式，每个 filter 模块在配置解析时都会把自己的过滤函数插入到链表头，同时内部又保存了原来的头节点。在过滤函数执行的最后只需要调用原头节点函数指针就可以让数据继续流到后面过滤模块，完成过滤链表的执行。

Nginx 进一步约定，模块里的变量 `next_filter` 必须是静态的，而且名字是 `ngx_http_next_header|body_filter`，从而简化了代码逻辑。每个模块里的变量名虽然相同，但因为声明为 `static`，所以不会互相干扰。^①

Nginx 的过滤链表应用了职责链模式 (chain of responsibility)，HTTP 响应数据顺序“流过”链表里的每一个模块，每个模块都可以对数据施加某种操作，前一个模块的输出作为后一个模块的输入，当数据走完整个职责链也就完成了处理。

职责链模式里链表的尾节点是关键，Nginx 定义了两个特殊的 filter 模块：`ngx_http_header_filter_module` 和 `ngx_http_write_filter_module`，它们分别是响应头链表和响应体链表的尾节点，负责最终的拼接响应头和数据发送工作。

这两个模块也同时负责初始化链表，保证最初的头节点指针不为空：

```
//位于文件 http/ngx_http_header_filter_module.c
static ngx_int_t
ngx_http_header_filter_init(ngx_conf_t *cf)    //postconfiguration 阶段调用
{
    ngx_http_top_header_filter = ngx_http_header_filter; //初始化链表头节点
    return NGX_OK;                                     //注意，没有 next 指针
}

//位于文件 http/ngx_http_write_filter_module.c
static ngx_int_t
ngx_http_write_filter_init(ngx_conf_t *cf)    //postconfiguration 阶段调用
{
    ngx_http_top_body_filter = ngx_http_write_filter; //初始化链表头节点
    return NGX_OK;                                     //注意，没有 next 指针
}
```

7.3.3 过滤函数的顺序

过滤链表是单向链表，没有可逆性和可交换性，所以链表里模块的执行顺序非常关键，错误的顺序可能会导致数据过滤后产生完全不同的结果。例如，如果链表里有 `gzip` 和 `gunzip` 两个模块，那么 `gzip=>...=>gunzip` 会得到一个非压缩数据，而 `gunzip=>...=>gzip`

^① 当然，我们也可以不遵守 Nginx 的约定，使用任意的变量名，也不一定是静态的。

则会得到一个压缩数据。

Nginx 的过滤链表虽然是一个链表，但它的构造发生在配置解析的 post-configuration 阶段，一旦构造完毕就不会改变，顺序在 Nginx 启动时就已经完全固定下来，运行时不能动态地增减。

在 ngx_http_block() 函数解析完配置文件后，会调用所有模块的 post-configuration 函数指针：

```
for (m = 0; cf->cycle->modules[m]; m++) { //顺序遍历模块 modules 数组
    module = cf->cycle->modules[m] ->ctx; //获取 http 模块的函数表

    if (module->postconfiguration) { //检查 postconfiguration 函数指针
        if (module->postconfiguration(cf) != NGX_OK) { //执行模块的初始化操作
            return NGX_CONF_ERROR;
        }
    }
}
```

从代码里可以看到，Nginx 是依据数组 cf->cycle->modules 里的顺序调用 post-configuration 函数指针的。所以按照单链表的添加方式，数组里位置在前的 filter 模块会在链表的尾部，而最后一个 filter 模块则是过滤链表的头节点，顺序与 cf->cycle->modules 数组里的正好相反。

Nginx 通过 configure 脚本，严格安排了每个 filter 模块的位置，保证 filter 模块可以正常工作，在生成的 objs/nginx_modules.c 里的初始 modules 数组是：

```
ngx_module_t *ngx_modules[] = {
    ... //core、event 等模块
    &ngx_http_write_filter_module, //最后一个 body filter 模块
    &ngx_http_header_filter_module, //最后一个 header filter 模块
    &ngx_http_chunked_filter_module,
    &ngx_http_range_header_filter_module,
    &ngx_http_gzip_filter_module,
    &ngx_http_postpone_filter_module,
    &ngx_http_ssi_filter_module,
    &ngx_http_charset_filter_module,
    &ngx_http_userid_filter_module,
    &ngx_http_headers_filter_module,

    ... //通常第三方 filter 模块在这里

    &ngx_http_copy_filter_module, //重要的数据链复制过滤模块
    &ngx_http_range_body_filter_module, //第一个 body filter 模块
}
```

```

ngx_http_not_modified_filter_module,           //第一个 header filter 模块
NULL
};

```

在开发 filter 模块时, config 脚本里要设置 `ngx_module_type=HTTP_FILTER/HTTP_AUX_FILTER`, 同时设置 `ngx_module_order`, 否则模块的位置错误会导致 Nginx 运行异常。

这些模块组成的过滤链表如图 7-5 所示。

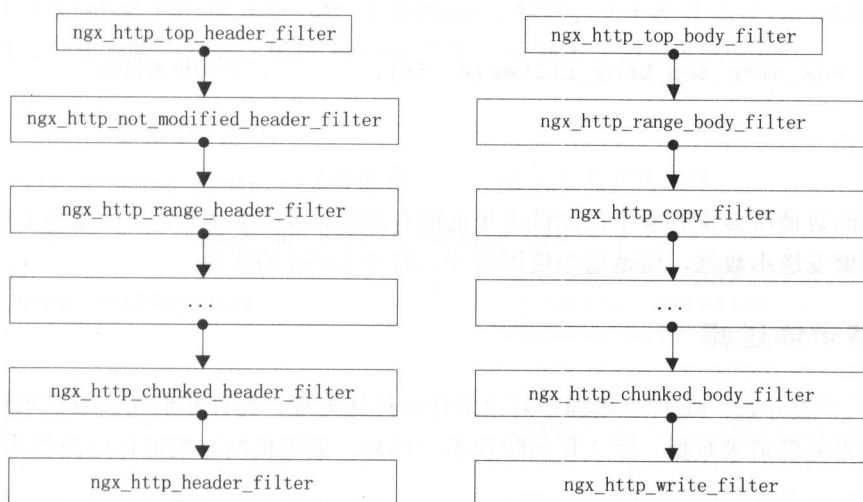


图 7-5 模块组成的过滤链表

7.3.4 过滤链表的运行机制

当 handler 模块生成了响应内容, 调用函数 `ngx_http_send_header()` 发送响应头时, 就会执行响应头过滤链表:

```

ngx_int_t
ngx_http_send_header(ngx_http_request_t *r)    //发送响应头
{
    if (r->header_sent) {                      //响应头发送标志位
        return NGX_ERROR;                     //已经发送则不再发送
    }

    if (r->err_status) {                       //检查请求的错误状态
        r->headers_out.status = r->err_status; //设置 HTTP 状态码
        r->headers_out.status_line.len = 0;
    }
}

```

```

    return ngx_http_top_header_filter(r);        //启动过滤链表
}

```

这样, HTTP 处理引擎就和过滤引擎成功地连接在了一起, 把响应内容传递给了过滤链表, 让各个 filter 模块执行数据的过滤处理, 最终发送给客户端。

响应体过滤链表的执行也基本类似, 它使用的函数是 `ngx_http_output_filter()`:

```

ngx_int_t
ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    rc = ngx_http_top_body_filter(r, in);        //启动过滤链表

    return rc;                                    //返回错误码
}

```

这两个函数调用会执行整个过滤链表里的所有函数, 可能成本较高, 所以出于效率考虑应尽量避免频繁发送小数据, 而是适当使用缓冲, 减少 send 的次数。

7.3.5 请求体过滤

从 1.8.0 版开始, Nginx 增加了对请求体的过滤处理, 允许我们在读取请求体时用过滤链表的方式来处理请求数据。除工作的阶段不一样外, 处理机制与响应体过滤基本类似。^①

请求体过滤函数的原型与 `ngx_http_output_body_filter_pt` 完全相同:

```

//定义在 http/ngx_http_core_module.h
typedef ngx_int_t (*ngx_http_request_body_filter_pt)
    (ngx_http_request_t *r, ngx_chain_t *chain);

```

同样的, 请求体过滤也有链表头节点:

```

//定义在 http/ngx_http.c
ngx_http_request_body_filter_pt  ngx_http_top_request_body_filter;

```

当 Nginx 读取请求体时, 会调用函数 `ngx_http_request_body_filter()`, 启动过滤链表。

^① Nginx 还没有提供请求头的过滤机制, 但我们完全可以在 `POST_READ` 阶段对请求头做处理。

7.4 源码分析

本节简要研究 `ngx_http_static_module` 和 `ngx_http_not_modified_filter_module` 这两个模块的源码，结合 7.2 节和 7.3 节来加深理解 Nginx 的 HTTP 处理框架，但仅研究与框架有关的注册和调用代码，并不解析内部的具体业务逻辑。

7.4.1 `ngx_http_static_module`

`ngx_http_static_module` 是一个 handler 模块，它工作在 CONTENT 阶段，读取磁盘上的静态文件作为响应内容，提供 Nginx 作为 Web 服务器最基本的功能。

模块定义

`ngx_http_static_module` 没有配置指令，模块定义非常简单：

```
ngx_http_module_t  ngx_http_static_module_ctx = {
    NULL,                                //preconfiguration
    ngx_http_static_init,                //postconfiguration
    NULL,                                //create main configuration
    NULL,                                //init main configuration
    NULL,                                //create server configuration
    NULL,                                //merge server configuration
    NULL,                                //create location configuration
    NULL,                                //merge location configuration
};
```

`ngx_http_static_module` 只使用了一个 `postconfiguration` 函数指针，用来注册模块的阶段处理函数。

注册处理函数

`ngx_http_static_init()` 访问 `ngx_http_core_module` 的 `phases` 数组，向 CONTENT 阶段添加 handler：

```
static ngx_int_t
ngx_http_static_init(ngx_conf_t *cf)
{
    cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);

    h = ngx_array_push(&cmcf->phases[NGX_HTTP_CONTENT_PHASE].handlers);
    *h = ngx_http_static_handler;           //添加数组元素
}
```

```

    return NGX_OK;
}

```

函数 `ngx_http_static_handler()` 是 `ngx_http_static_module` 的处理函数，它检查 URI 的有效性，映射 URI 到磁盘路径，再访问文件，最后调用 `ngx_http_output_filter()`，把文件内容交给过滤链表处理。

7.4.2 ngx_http_not_modified_filter_module

`ngx_http_not_modified_filter_module` 是一个 header filter 模块，是 `ngx_module` 数组里的最后一个元素，同时也是过滤链表的第一个节点。它处理响应头，检查 If-Modified-Since 头，决定是否返回 304 状态码。

模块定义

`ngx_http_static_module` 没有配置指令，模块定义也同样简单：

```

static ngx_http_module_t  ngx_http_not_modified_filter_module_ctx = {
    NULL,                                     //preconfiguration
    ngx_http_not_modified_filter_init,       //postconfiguration
    NULL,                                     //create main configuration
    NULL,                                     //init main configuration
    NULL,                                     //create server configuration
    NULL,                                     //merge server configuration
    NULL,                                     //create location configuration
    NULL,                                     //merge location configuration
};

```

注册处理函数

`ngx_http_not_modified_filter_init()` 操作 Nginx 过滤链表头节点，注册自己的处理函数：

```

//静态变量，存储 next 指针
static ngx_http_output_header_filter_pt  ngx_http_next_header_filter;

static ngx_int_t
ngx_http_not_modified_filter_init(ngx_conf_t *cf)
{
    //向链表添加节点
    ngx_http_next_header_filter = ngx_http_top_header_filter;
    ngx_http_top_header_filter = ngx_http_not_modified_header_filter;

    return NGX_OK;
}

```

`ngx_http_not_modified_header_filter()` 是模块的处理函数，它检查请求头。如果文件没有被修改，就设置 HTTP 状态码为 `NGX_HTTP_NOT_MODIFIED` (304)，然后调用 `ngx_http_next_header_filter` 执行后续的过滤模块。

7.5 C++封装

本章的前几节已经比较详尽地分析了 Nginx 的 HTTP 框架和运行机制，现在我们要使用 C++ 来实现对它们的封装。

7.5.1 NgxModuleCtx

模块的环境数据存储在全局 `ngx_http_request_t` 的 `ctx` 数组里，形式上与模块的配置数据存储的 `main_conf/srv_conf/loc_conf` 相同，所以可以用与 6.4 节类似的方法来实现 C++ 类，提供便捷的访问方式。

类定义

`NgxModuleCtx` 与 `NgxModuleConfig` 类似，保存模块的 `ctx_index` 作为内部成员：

```
class NgxModuleCtx final
{
public:
    NgxModuleCtx(ngx_uint_t idx): m_idx(idx)           //获取模块的序号
    {}
    ~NgxModuleCtx() = default;
private:
    ngx_uint_t m_idx = 0;                             //保存模块的序号
public:
    ngx_uint_t index() const                           //获取模块的序号
    {
        return m_idx;                                 //模块的序号
    }
    ...                                                //其他成员函数，见后
};
```

访问环境数据

模仿宏 `ngx_http_get_module_ctx`，使用 `index()` 可以得到模块存储的环境数据：

```
private:
    typedef void* raw_pointer;                        //简化类型定义
    raw_pointer& ctx(ngx_http_request_t* r) const    //返回 void*&类型
    {
```

```
    return r->ctx[index()];           //访问 ctx 数组
}
```

注意: `ctx()` 成员函数返回的不是简单的 `void*` 指针, 而是 `void*` 的引用形式, 所以我们可以把它当作左值直接操作。

操作函数

`ctx()` 成员函数实际上就是 `ngx_http_request_t::ctx` 数组里存储的指针, 所以使用它可以操作任意模块的环境数据:

```
public:
    bool empty(ngx_http_request_t* r) const           //是否有环境数据
    {
        return !ctx(r);                               //指针为空则不存在
    }

    void clear(ngx_http_request_t* r) const           //清空环境数据
    {
        ctx(r) = nullptr;                             //左值, 直接置为空指针
    }
```

模板成员函数 `data()` 检查模块的环境数据是否存在, 如果没有就直接构造并返回: ①

```
template<typename T>                               //模板参数指明数据类型
T& data(ngx_http_request_t* r) const               //获取模块的环境数据
{
    if(empty(r))                                   //检查是否存在
    {
        ctx(r) = NgxPool(r).alloc<T>();           //从内存池创建对象
    }

    return *reinterpret_cast<T*>(ctx(r));         //返回转型后的环境数据
}
```

注意 `data()` 函数返回的是对象的引用, 而不是指针。

NgxModule 工厂类

6.4.2 节 `NgxModule` 类不仅可以生产 `NgxModuleConfig` 对象, 还可以生产 `NgxModuleCtx`:

```
template<...>
```

① GitHub 上的代码有所增强, 使用 `boost::enable_if` 允许 `ctx` 以内存池作为参数构造。


```

class NgxModule                                     //模块的代理类
{
public:
    NgxModule(ngx_module_t& m):                     //引用模块对象
        m_conf(m.ctx_index), m_ctx(m.ctx_index)    //使用序号 ctx_index
    {}
private:
    typedef NgxModuleCtx ctx_type;                  //简化类型定义
    ctx_type      m_ctx;                            //NgxModuleCtx 对象
public:
    const ctx_type& ctx() const                     //产生 NgxModuleCtx 对象
    {
        return m_ctx;                              //返回成员的引用
    }

    template<typename T>
    T& data(ngx_http_request_t* r) const             //直接获取环境数据
    {
        return ctx().template data<T>(r);           //注意模板函数的调用方式
    }
};

```

同样的，我们定义宏 `NGX_MOD_CTX_INSTANCE` 来简化使用：

```

#define NGX_MOD_CTX_INSTANCE(T, mod, c, ... ) \    //方便使用的模块定义宏
struct T { \                                       //模块类定义
    ...
    static const ctx_type& ctx() \                //获取 ctx 结构体
    { return instance().ctx(); } \
    static ctx_data_type& data(mod_type::ngx_session_type* r) \ //ctx 数据
    { return instance().template data<ctx_data_type>(r); } \
};

```

7.5.2 NgxHttpCoreModule

模块 `ngx_http_core_module` 是 Nginx HTTP 框架的核心，它的 `main/server/location` 三个层次的配置结构都非常重要，在编写 http 模块时需要频繁访问，所以有必要从 `NgxModule` 派生出一个专门的 `NgxHttpCoreModule` 类来代理它。

`NgxHttpCoreModule` 继承了 `NgxModule` 获取配置结构的能力，然后增加本模块独有的注册 HTTP 处理函数的功能。

类定义

`NgxHttpCoreModule` 是 `NgxModule` 的子类，因为它有三个不同的配置结构，所以

模板参数列表略长:

```
class NgxHttpCoreModule final : //final 禁止继承
    public NgxModule<ngx_http_core_loc_conf_t, //三个配置结构类型
                ngx_http_core_srv_conf_t, //注意顺序
                ngx_http_core_main_conf_t >
{
public:
    typedef NgxModule<...> super_type; //父类定义
    typedef NgxHttpCoreModule this_type; //自身类型定义
public:
    NgxHttpCoreModule() : super_type(ngx_http_core_module) //初始化父类
    {}
    ~NgxHttpCoreModule() = default;
    ... //其他成员函数，见后
};
```

单件访问点

NgxHttpCoreModule 需要自己实现单件访问函数 instance():

```
public:
    static NgxHttpCoreModule& instance() //单件访问函数
    {
        static NgxHttpCoreModule m; //不需要额外的参数构造
        return m;
    }
```

注册处理函数

Nginx 提供了两种注册 HTTP 处理函数的方法:

- 在配置文件解析完毕之后，也就是 postconfiguration 函数指针里，操作 ngx_http_core_main_conf_t 结构，向 phases 数组添加元素，就可以向框架注册应用于任何 HTTP 请求的阶段处理函数。
- 在配置解析到某个 location 内部指令时，设置 ngx_http_core_loc_conf_t 结构的 handler 成员，可以注册仅用于该 location 的处理函数。

使用 C++ 的函数重载特性，我们可以把这两个注册动作整合在一个同名函数里:

```
public:
    template<typename F>
    static void handler(ngx_conf_t* cf, F f, //设置阶段处理 handler
                      ngx_http_phases p) const //使用阶段参数
    {
```

```

auto& c = instance().conf().main(cf);           //获取 main 配置

typedef ngx_array_t<ngx_http_handler_pt> handler_array_t;

handler_array_t arr(c.phases[p].handlers);      //获取 phases 数组
arr.push(f);                                    //设置 handler
}

template<typename F>
static void handler(ngx_conf_t* cf, F f) const  //设置内容处理 handler
{
    instance().conf().loc(cf).handler = f; //获取 location 配置并设置 handler
}

```

用法

`NgxHttpCoreModule` 类把访问配置结构数组和处理函数的操作都封装了起来，所以注册 handler 的工作变得非常简单和易于理解，例如：

```

NgxHttpCoreModule::handler(                //使用 NgxHttpCoreModule 单件
    cf, func, NGX_HTTP_REWRITE_PHASE);      //添加 rewrite 阶段的 handler

NgxHttpCoreModule::handler(cf, func);      //添加内容 handler

```

7.5.3 NgxFilter

Nginx 过滤机制要求每个模块使用内部的私有静态变量来保存 next 指针，使用类的静态变量可以达到同样的效果。由于 C++11 还不支持模板变量，所以本书采用模板函数来模拟实现。^①

类定义

`NgxFilter` 是一个模板类，定义如下：

```

template<typename Tag>                                //Tag 模板参数供实例化用
class NgxFilter final
{
public:
    typedef ngx_http_output_header_filter_pt  header_filter_ptr;
    typedef ngx_http_output_body_filter_pt    body_filter_ptr;
public:
    NgxFilter() = default;

```

^① `NgxFilter` 暂不支持请求体过滤链表，但可以很容易地添加支持，读者可参考 GitHub 资源。

```

~NgxFilter() = default;
private:
    template<typename T> //模板参数确定变量的类型
    static T& next()      //静态成员函数
    {
        static T next_filter; //定义 next 指针
        return next_filter;
    }
    ... //其他成员函数，见后
};

```

静态成员函数 `next()` 的行为很像变量，它内部声明了一个变量，但类型需要由模板参数指定，所以只要变动模板参数 `T` 就可以得到对应类型的静态变量，行为很像 C++14 里的模板变量。

`NgxFilter` 巧妙地利用了模板参数 `Tag`，它的作用仅仅是一个编译期的标记，实例化后可以生成唯一的模板实例类，这样每个 `NgxFilter<Tag>` 类都可以拥有只属于自己的类静态变量，达到了与 Nginx 文件作用域静态变量同样的效果。

注册过滤链表

使用成员函数 `next()` 就可以操作 Nginx 的链表指针：

```

private:
    template<typename T> //T 是函数指针类型
    static void set(T& top, T p) //向过滤链表添加头节点
    {
        next<T>() = top; //保存头节点指针
        top = p; //设置头节点指针
    }

```

`init()` 函数的参数确定模块的过滤函数指针。如果指针不是空指针，那么就可以调用 `set()` 向过滤器链表添加头节点：

```

public:
    static void init(header_filter_ptr header_filter,
                     body_filter_ptr body_filter)
    {
        if(header_filter) //header_filter 指针
        {
            set(ngx_http_top_header_filter, header_filter);
        }

        if(body_filter) //body_filter 指针
        {

```

```

        set(ngx_http_top_body_filter, body_filter);
    }
}

```

在这里我们利用了 C++ 函数自动推导模板参数的能力，实现了 `set()` 函数的模板实例化，最终访问 `next()` 里正确类型的静态变量。

调用后续过滤链表

成员函数 `next()` 存储了后续过滤链表指针，所以能够直接调用：

```

public:
    static ngx_int_t next(ngx_http_request_t* r)    //调用 header 过滤链表
    {
        return next<header_filter_ptr>()(r);      //显式模板实例化
    }

    static ngx_int_t next(                        //重载调用 body 过滤链表
        ngx_http_request_t* r, ngx_chain_t *chain)
    {
        return next<body_filter_ptr>()(r, chain);  //显式模板实例化
    }

```

在开发 `filter` 模块时，我们需要调用这两个 `next()` 函数，执行后续的过滤链表。

用法

使用 `NgxFILTER` 必须要提供 `Tag` 模板参数，实例化出一个专用的过滤器类，例如：

```
typedef NgxFILTER<tag>    this_filter;
```

然后在 `postconfiguration` 阶段就可以调用 `init()` 函数，注册过滤处理函数。

在第 8 章我们会使用 `NgxFILTER` 开发一个 `filter` 模块，这里不再列出更多的示范代码。

7.6 总结

本章详细研究了 Nginx 的 HTTP 处理框架，解析了两类主要的 http 模块——handler 模块和 filter 模块的工作原理，它们是 Nginx 宫殿里最光彩夺目的那些厅堂。

所有 Web 服务器对 HTTP 请求的基本处理流程都是类似的，但 Nginx 对请求处理的阶段划分则再次体现了作者 Igor Sysoev 的精妙设计思想。

接收完请求头数据后，Nginx 就进入了阶段式处理引擎。Nginx 把 HTTP 请求处理划分

为 access、rewrite、content、log 等 11 个阶段（但有 4 个阶段仅供框架内部使用），相当于在处理流程里提供了 11 个 hook 点，模块可以自由选择自己感兴趣的阶段，注册自己的处理函数，实现地址改写、访问控制、产生内容、记录日志等操作。

处理引擎产生响应内容后，Nginx 会进入过滤处理阶段，可以把这个阶段理解为在 content 和 log 之间的一个特殊的“filter”阶段。在这里 Nginx 把 filter 模块组织成了一个链表，响应头和响应体流过这个链表进行各种加工处理，最后才真正发送给客户端。

handler 模块和 filter 模块本质上都是职责链模式，请求被链里的每一个模块所处理，只是 handler 模块的流程更加复杂，因为存在 URI 改写，所以在引擎链表里可以反复跳转。

Nginx 使用 configure 脚本生成 ngx_modules 数组，它在确定了 http 的模块正确执行顺序，运行时不能动态调整。在 Nginx 启动时框架顺序调用模块的 postconfiguration 函数指针，完成初始化，之后模块就会各就各位，各司其职，分工协作，在引擎函数的驱动下共同完成 HTTP 请求的处理。

开发 handler 模块需要实现 ngx_http_handler_pt 函数，Nginx 提供了两种注册方式。一种是 phase handler，操作 ngx_http_core_main_conf_t 结构的 phases 数组；另一种是 content handler，操作 ngx_http_core_loc_conf_t 结构的 handler 成员。第一种方式是最通用的方式，可以挂载到任意处理阶段，处理所有的 HTTP 请求；而第二种方式则只对某个具体的 location 生效，而且只能用来产生响应内容。

开发 filter 模块需要实现 ngx_http_output_header_filter_pt 或 ngx_http_output_body_filter_pt 函数，分别过滤处理响应头和响应体。函数的注册要比 handler 模块简单一些，只要定义一个 next 指针就可以加入过滤链表。

handler 模块和 filter 模块都可以实现处理 HTTP 请求的功能，在实践开发时我们要对这两者的应用范围和工作机制有正确的认识，选择合适的模块类型。handler 模块的应用范围最广，不仅可以生成响应内容，还可以执行权限检查、请求重写等其他功能。而 filter 模块则不负责生产数据，它只能加工数据。在编写生成响应内容的 handler 模块时应该尽量让模块的工作简单，只产生原始数据，把对数据的复杂处理逻辑实现在另外的 filter 模块里，多个 filter 模块协作完成处理。

本章的最后我们使用 C++封装了 Nginx HTTP 框架里三个非常重要且基本的功能：访问环境数据，注册处理函数和操作过滤链表，使用这些 C++工具类可以很好地辅助开发 http 模块，简化实现代码。

第 8 章

Nginx HTTP请求处理

本章将详细解析 Nginx 处理 HTTP 请求的核心结构体 `ngx_http_request_t`，阐述其中的请求头、请求体、响应头、响应体等 HTTP 处理的关键数据成员和操作方法，并提供易用的 C++ 高级接口，读者可以学习到开发可用的 handler 模块和 filter 模块的知识。

8.1 状态码

Nginx 用宏定义了常用的 HTTP 状态码，包括常见的 200、302、403 等，以及一些 Nginx 自己特有的状态码。

下面仅列出了部分宏，完全的定义可阅读头文件 `<http/ngx_http_request.h>`:

```
// 定义在 http/ngx_http_request.h
#define NGX_HTTP_OK 200
#define NGX_HTTP_CREATED 201
#define NGX_HTTP_ACCEPTED 202
... //其他 2xx 状态码

#define NGX_HTTP_SPECIAL_RESPONSE 300
#define NGX_HTTP_MOVED_PERMANENTLY 301
#define NGX_HTTP_MOVED_TEMPORARILY 302
... //其他 3xx 状态码

#define NGX_HTTP_BAD_REQUEST 400
#define NGX_HTTP_UNAUTHORIZED 401
#define NGX_HTTP_FORBIDDEN 403
#define NGX_HTTP_NOT_FOUND 404
... //其他 4xx 状态码

#define NGX_HTTP_INTERNAL_SERVER_ERROR 500
```

```

#define NGX_HTTP_NOT_IMPLEMENTED          501
#define NGX_HTTP_BAD_GATEWAY              502
...                                         //其他 5XX 状态码

```

在编写 HTTP 模块时应当尽量使用这些含义明确的 Nginx 预定义宏，当然，直接使用数字也是可以的。

8.2 请求结构体

之前我们研究 Nginx 的 HTTP 框架时都用到了结构体 `ngx_http_request_t`，它包含了 Nginx 处理请求过程中需要的所有信息，处理 HTTP 请求实际上就是操作 `ngx_http_request_t` 里的各个成员。

本章将介绍 `ngx_http_request_t` 中与 HTTP 协议 (RFC2616)^① 有关的部分，它的定义摘要如下：

```

// 定义在 http/ngx_http_request.h
struct ngx_http_request_s {
    void**                ctx;                //处理请求的环境数据
    void**                main_conf;          //main 层次的配置信息数组
    void**                srv_conf;          //server 层次的配置信息数组
    void**                loc_conf;          //location 层次的配置信息数组

    ngx_pool_t*           pool;              //请求使用的内存池

    ngx_buf_t*            header_in;         //接收数据的缓冲区

    ngx_http_headers_in_t headers_in;        //请求头数据结构
    ngx_http_headers_out_t headers_out;      //响应头数据结构

    ngx_http_request_body_t* request_body;   //请求体数据结构

    ngx_uint_t            method;            //请求方法
    ngx_uint_t            http_version;      //HTTP 版本号

    ngx_str_t             request_line;      //存储完整的请求行字符串
    ngx_str_t             uri;              //请求的 uri
    ngx_str_t             args;             //请求的参数
    ngx_str_t             exten;            //请求的扩展名
    ngx_str_t             unparsed_uri;     //未解析的原始 uri

```

① 请读者参考 <http://www.ietf.org/rfc/rfc2616.txt>。


```

ngx_str_t      method_name;      //请求方法名
ngx_str_t      http_protocol;    //HTTP 协议版本字符串

unsigned        uri_changed:1;    //uri 是否已经被改写
unsigned        header_only:1;   //是否只有头
unsigned        discard_body:1;  //是否丢弃请求体
unsigned        reading_body:1;  //是否正在读取请求体
unsigned        header_sent:1;   //是否已经发送了响应头
unsigned        allow_ranges:1;  //是否支持 range 协议

... //其他成员
};

```

本章之后的小节将结合 HTTP 协议把它分解为请求行、请求头、请求体等部分逐步研究。

8.3 请求行

HTTP 请求行包括请求方法，请求的 URI、HTTP 版本等信息，在 RFC2616 里的定义是：

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

一个请求行的例子如下：

```
GET /index.html HTTP/1.1
```

其中的“GET”是请求方法，“/index.html”是 URI，也就是通常所说的 URL 或请求地址，“HTTP/1.1”是协议的版本。

Nginx 在接收到客户端发来的 HTTP 请求后，会调用函数 `ngx_http_process_request_line()` 解析请求行数据，存储到 `ngx_http_request_t::request_line` 成员，然后再进一步分解出更详细的信息（参见第 17 章）。

8.3.1 请求方法

`ngx_http_request_t` 使用两个成员存储 HTTP 请求方法（method）信息：

```

ngx_uint_t      method;          //方法的整数表示
ngx_str_t       method_name;     //方法的字符串表示

```

`method` 是 Nginx 解析请求行后得到的方法标识，是一个整数，`method_name` 则是方法的字符串表示，例如 GET/HEAD/PUT/POST/COPY 等，相当于变量 `$request_method`。

Nginx 使用宏定义了 HTTP 协议里已有的方法：

```
#define NGX_HTTP_UNKNOWN    0x0001    //错误方法
```

```

#define NGX_HTTP_GET          0x0002          //GET
#define NGX_HTTP_HEAD         0x0004          //HEAD
#define NGX_HTTP_POST         0x0008          //POST
#define NGX_HTTP_PUT          0x0010          //PUT
#define NGX_HTTP_DELETE       0x0020          //DELETE
...                                     //其他方法值

```

在编写 http 模块时应当总使用 method 成员与这些宏进行比较来确定请求方法, 尽量避免使用字符串形式的 method_name, 因为整数的比较操作更快, method_name 通常只用来记录日志或者调试。

检查 method 的一个小技巧是使用逻辑或操作符 (&), 可以一次检查多个值, 例如:

```
if (r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD)) //要求是 get 或 head 方法
```

8.3.2 协议版本号

成员变量 http_version 标记了 HTTP 请求的版本:

目前 HTTP 协议有 0.9、1.0、1.1 和 2.0 四个版本, Nginx 定义四个宏与它们对应: ①

```

#define NGX_HTTP_VERSION_9          9
#define NGX_HTTP_VERSION_10         1000
#define NGX_HTTP_VERSION_11         1001
#define NGX_HTTP_VERSION_20         2000

```

判断协议的版本号只需直接比较 http_version 成员和这四个宏即可。

成员 http_protocol 存储的是请求行里完整的版本字符串, 但因为 http 模块处理的协议必定是 HTTP, 所以它的用处不大。

8.3.3 资源标识符

ngx_http_request_t 里有四个成员描述请求里的资源标识符 (URI):

```

ngx_str_t      uri;           //不带参数的 uri, 即 $uri
ngx_str_t      args;         //uri 里的参数, 即 $args
ngx_str_t      exten;        //uri 里的文件扩展名
ngx_str_t      unparsed_uri; //未解码的原始 uri 字符串, 带参数

```

成员 uri 和 unparsed_uri 都表示请求行里的 URI 字符串, 两者的区别是 uri 做了解码, 且不含有 “?” 后面的参数, 即 \$uri; 而 unparsed_uri 是原始的未经任何处理的 URI

① 2015 年 HTTP 2.0 协议正式发布, Nginx 不再支持 Google 的 SPDY 协议。

字符串，包含参数，它相当于\$request_uri。

成员 args 表示 URI 里的参数，即 “?” 后面的字符串\$args，不包含 “?”，它也没有经过 URL 解码。

成员 exten 表示 uri 里的文件扩展名，如果文件没有扩展名，那么它就是个空字符串。

例如，如果有一个请求：

```
curl -v http://localhost/test/%40index.html?a=42
```

那么就有：

```
uri          = /test/@index.html
args         = a=42
exten        = html
unparsed_uri = /test/%40index.html?a=42
```

在 HTTP 请求处理的任意阶段可以使用下面两个函数实现 URI 的跳转：

```
ngx_int_t ngx_http_named_location(ngx_http_request_t *r, ngx_str_t *name);
ngx_int_t ngx_http_internal_redirect(ngx_http_request_t *r,
                                     ngx_str_t *uri, ngx_str_t *args);
```

两个函数的区别是前者只能跳转到以 “@” 开头的 location，不能传递参数；而后者更加灵活，可以跳转到本 server 块内部的任意 location。

Nginx 还提供一个函数 ngx_http_parse_unsafe_uri()，它可以检查 uri 是否是一个不安全的 uri（例如含有 “../”），声明是：

```
ngx_int_t ngx_http_parse_unsafe_uri(
    ngx_http_request_t *r, ngx_str_t *uri, ngx_str_t *args, ngx_uint_t *flags);
```

8.4 请求头

ngx_http_request_t 用两个成员存储收到的 HTTP 头部信息：

```
ngx_buf_t*          header_in;           //接收数据的缓冲区
ngx_http_headers_in_t headers_in;        //解析后的头部信息
```

header_in 是一个缓冲区块，存放了 Nginx 收到的 HTTP 原始请求数据，我们通常不需要关心，而是直接使用已经解析好的 headers_in 成员。

headers_in 的类型是 ngx_http_headers_in_t，定义摘要如下：

```
typedef struct {
```

```

ngx_list_t          headers;           //头信息链表

ngx_table_elt_t*    host;              //host 头
ngx_table_elt_t*    connection;       //connection 头
ngx_table_elt_t*    if_modified_since; //if_modified_since 头
ngx_table_elt_t*    user_agent;       //user_agent 头
ngx_table_elt_t*    content_length;   //content_length 头
ngx_table_elt_t*    content_type;     //content_type 头
ngx_table_elt_t*    range;            //range 头
ngx_table_elt_t*    if_range;         //if_range 头
ngx_table_elt_t*    keep_alive;       //keep_alive 头

off_t               content_length_n;  //content_length 头的整数形式
time_t              keep_alive_n;     //keep_alive 头的整数形式

...

//其他成员

} ngx_http_headers_in_t;

```

`ngx_http_headers_in_t` 结构体初看上去很复杂,但实际上很容易掌握。它的核心成员是 `headers`,以链表的方式存储了所有的请求头,相当于 `NgxList<ngx_table_elt_t>`,我们可以用 4.2 节里介绍的各种方法来操作它。

`ngx_http_headers_in_t` 里的其他成员大多数是 `ngx_table_elt_t*` 指针,指向 `headers` 链表里的元素,为常用的头信息访问提供了快捷方式,无须遍历链表就可以直接获取头信息。如果指针是 `nullptr`,就表示没有该头部信息。

对于 `content_length` 和 `keep_alive` 这两个头,`ngx_http_headers_in_t` 还提供了整数形式的成员 `content_length_n` 和 `keep_alive_n`,免去了我们自己解析的麻烦。

8.5 请求体

`ngx_http_request_t` 用成员 `request_body` 存储收到的 HTTP 请求体数据。^①

8.5.1 结构定义

`ngx_http_request_body_t` 的定义如下:

```
typedef struct {
```

① 本书不讨论 Nginx 临时文件存储请求体的方式,所以需要用指令“`client_body_buffer_size`”保证缓冲区足够大,内存可以容纳数据。

```

ngx_temp_file_t*      temp_file;      //存放请求体的临时文件
ngx_chain_t*          bufs;           //存放请求体的内存区
ngx_buf_t*            buf;           //接收缓冲区
off_t                 rest;          //待接收的剩余字节数
ngx_chain_t*          free;
ngx_chain_t*          busy;
ngx_http_chunked_t*    chunked;
ngx_http_client_body_handler_pt post_handler; //接收完数据后的回调函数
} ngx_http_request_body_t;

```

`ngx_http_request_body_t` 里的成员较多，大多数是由 Nginx 内部使用（相当于 private 成员），用于异步读取数据。我们通常只需关心成员 `bufs`，它存储了收到的请求体数据。

8.5.2 操作函数

很多时候请求体是可有可无的（例如最常用的 GET），所以可以直接“丢弃”请求体：

```

//定义在 http/ngx_http.h
ngx_int_t ngx_http_discard_request_body(ngx_http_request_t *r);

```

调用函数 `ngx_http_discard_request_body()` 会让 Nginx 继续在连接上接收数据，但并不会存入 `request_body`。

如果确实需要获取请求体数据，那么就需要使用 Nginx 提供的异步读取请求体函数 `ngx_http_read_client_request_body()`，它的声明是：

```

//回调函数，定义在 http/ngx_http_request.h
typedef void (*ngx_http_client_body_handler_pt)(ngx_http_request_t *r);

```

```

//定义在 http/ngx_http.h
ngx_int_t ngx_http_read_client_request_body(ngx_http_request_t *r,
                                             ngx_http_client_body_handler_pt post_handler);

```

`ngx_http_read_client_request_body()` 在调用后通常会立即返回，当 Nginx 读取数据后会回调函数 `post_handler`，用户可以在这个函数里处理收到的请求体数据。

8.6 响应头

Nginx 使用结构体 `ngx_http_headers_out_t` 表示响应头，它包含了 HTTP 响应里的状态行和头部信息，设计上与 `ngx_http_headers_in_t` 结构并不对称（这一点比较奇怪）。

8.6.1 结构定义

`ngx_http_headers_out_t` 的定义摘要如下：

```
typedef struct {
    ngx_list_t                headers;           //头信息链表

    ngx_uint_t                status;           //状态码
    ngx_str_t                 status_line;       //状态行

    ngx_table_elt_t*          server;
    ngx_table_elt_t*          date;
    ngx_table_elt_t*          content_length;
    ngx_table_elt_t*          expires;
    ngx_table_elt_t*          etag;

    ngx_str_t                 content_type;

    off_t                     content_length_n; //响应数据长度
    time_t                    date_time;
    time_t                    last_modified_time;

    ...                                     //其他成员
} ngx_http_headers_out_t;
```

`ngx_http_headers_out_t` 结构体与 `ngx_http_headers_in_t` 很类似，核心成员仍然是链表 `headers`，用于存储响应头信息。我们也可以直接用 `server`、`date` 等指针直接指定标准头，不一定非要在链表里。

成员 `status` 是响应状态行里的状态码，它的取值为 8.1 节里定义的宏，如果正确处理了 HTTP 请求，值就应该是 `NGX_HTTP_OK`。

还有一个比较重要的成员是 `content_length_n`，用于设置响应体数据的长度，Nginx 会自动把整数形式的 `content_length_n` 转换为响应头里的字符串形式。

8.6.2 操作函数

响应头均是以 `ngx_table_elt_t` 的形式存储，如果想要删除某个头信息，只需把头对应的 `ngx_table_elt_t` 的 `hash` 值置为 0 就可以了，例如：

```
r->headers_out.content_length->hash = 0;           //散列值清空
r->headers_out.content_length = nullptr;           //指针也要置空，更加安全
```

Nginx 还定义了几个函数宏，可以便捷地清空常用的响应头信息：

```
//定义在头文件 http/nginx_http_core_module.h
#define ngx_http_clear_content_length(r)
#define ngx_http_clear_accept_ranges(r)
#define ngx_http_clear_last_modified(r)
#define ngx_http_clear_location(r)
#define ngx_http_clear_etag(r)
```

函数 `ngx_http_send_header()` 可以把响应头发送给客户端，内部会调用 `ngx_http_top_header_filter`，让头信息经过过滤链表最终发送出去（我们已经在 7.3.4 节看到了它的实现源码）：

```
ngx_int_t ngx_http_send_header(ngx_http_request_t *r);
```

另一个函数 `ngx_http_clean_header()` 可以清除所有的响应头信息：

```
void ngx_http_clean_header(ngx_http_request_t *r);
```

8.7 响应体

Nginx 里并没有为响应体定义专门的数据结构，实际上，响应体就是一些数据，可以用 `ngx_buf_t` 和 `ngx_chain_t` 来表示。

调用函数 `ngx_http_output_filter()` 可以把响应体数据经由过滤链表发给客户端：

```
ngx_int_t ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in);
```

必须要注意的是，Nginx 采用异步、缓存的方式处理数据，函数调用时不一定会立即使用链表里的数据，数据必须在调用后的一段时间里依然可用，所以最好是在 `r->pool` 里分配待发送的内存存储数据，否则可能会发生严重的运行时错误。

函数 `ngx_http_send_special()` 可以发送用于控制的特殊 `ngx_buf_t` 对象：

```
#define NGX_HTTP_LAST 1 //设置为最后一个缓冲区，即 eof
#define NGX_HTTP_FLUSH 2 //刷新缓冲区
ngx_int_t ngx_http_send_special(ngx_http_request_t *r, ngx_uint_t flags);
```

8.8 源码分析

本节继续 7.4 节的讨论，研究 `ngx_http_static_module` 和 `ngx_http_not_modified_filter_module` 这两个模块的源码，解析具体业务逻辑，帮助读者进一步了解 Nginx 的 HTTP 处理功能。

8.8.1 ngx_http_static_module

ngx_http_static_module 工作在 CONTENT 阶段, 读取磁盘上的静态文件, 核心处理函数是 ngx_http_static_handler, 这里只列出其中的部分关键代码。

ngx_http_static_handler 首先检查 URI 的有效性, 然后把 URI 映射到磁盘上的实际文件路径:

```

if (!(r->method &
    (NGX_HTTP_GET|NGX_HTTP_HEAD|NGX_HTTP_POST))) { //使用逻辑与检查 method //必须是 GET/HEAD/POST
    return NGX_HTTP_NOT_ALLOWED;
}

if (r->uri.data[r->uri.len - 1] == '/') { //检查 uri, 最后一个字符不能是 '/'
    return NGX_DECLINED; //也就是说不允许操作目录
}

last = ngx_http_map_uri_to_path( //把 uri 映射到磁盘目录 path
    r, &path, &root, 0); //会根据 root/alias 指令决定正确的位置

path.len = last - path.data; //last 是计算后的末尾位置, 减去开始地址就是 path 长度

clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

ngx_memzero(&of, sizeof(ngx_open_file_info_t)); //设置打开文件的选项

if (ngx_open_cached_file( //打开位于 path 上的文件, 使用选项 of
    clcf->open_file_cache, &path, &of, r->pool) != NGX_OK)
{ ... }

if (of.is_dir) //处理文件是目录的情况
{ ... }

文件成功打开后, ngx_http_static_handler 丢弃请求体, 准备响应头

if (r->method == NGX_HTTP_POST) { //不允许 POST 数据到文件, 即只允许 GET/HEAD
    return NGX_HTTP_NOT_ALLOWED;
}

rc = ngx_http_discard_request_body(r); //丢弃请求体

r->headers_out.status = NGX_HTTP_OK; //设置响应头信息: 状态码、长度、修改时间
r->headers_out.content_length_n = of.size;
r->headers_out.last_modified_time = of.mtime;

r->allow_ranges = 1; //允许 range 请求

```


最后是分配缓冲区，发送响应体数据：

```
b = ngx_palloc(r->pool, sizeof(ngx_buf_t)); //创建缓冲区结构体，不分配实际内存
b->file = ngx_palloc(r->pool, sizeof(ngx_file_t)); //创建文件结构体，用于描述磁盘文件

rc = ngx_http_send_header(r); //先发送头

if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) { //发送出错
    //或者是 HEAD 请求，就无须再发送 body
    return rc;
}

b->file_pos = 0; //设置缓冲区里的 file 位置信息
b->file_last = of.size;

b->last_buf = (r == r->main) ? 1: 0; //如果是主请求则设置 last_buf，即最后一块
b->last_in_chain = 1; //因为只发送一个文件，所以必定是链里的最后一个

out.buf = b; //把缓冲区放进链表里
out.next = NULL; //重要操作，链表的指针必须是 nullptr，否则会发生严重错误

return ngx_http_output_filter(r, &out); //调用过滤链表，最终发送给客户端
```

8.8.2 ngx_http_not_modified_filter_module

ngx_http_not_modified_filter_module 是 Nginx 过滤链表里的第一个模块，它检查 If-Modified-Since 和 If-None-Match 头，如果文件没有变动就返回 304。

模块的主要处理函数是 ngx_http_not_modified_header_filter，实现摘要如下：

```
if (r->headers_out.status != NGX_HTTP_OK //请求处理失败
    || r != r->main //不是主请求
    || r->disable_not_modified) //禁用 not_modified
{
    return ngx_http_next_header_filter(r); //不需要其他操作，直接走后续链表处理
}

if (r->headers_in.if_modified_since || //检查是否有 if_modified_since
    r->headers_in.if_none_match) { //和 if_none_match 头

    if (r->headers_in.if_modified_since //有 if_modified_since
        && ngx_http_test_if_modified(r)) //且判断时间戳已经变动
    {
        return ngx_http_next_header_filter(r); //不发 304，走后续链表处理
    }
}
```

```

if (r->headers_in.if_none_match          //有 if_none_match 头
    && !ngx_http_test_if_match(...))      //判断 etag 有变动
{
    return ngx_http_next_header_filter(r); //不发 304，走后续链表处理
}

```

//前两处测试完毕，文件没有任何变动，可以返回 304 状态码，要修改响应头

```

r->headers_out.status = NGX_HTTP_NOT_MODIFIED; //把状态码改为 304
r->headers_out.status_line.len = 0;            //状态行清空
r->headers_out.content_type.len = 0;           //内容清空
ngx_http_clear_content_length(r);             //内容长度清零
ngx_http_clear_accept_ranges(r);              //清除 accept_range 标志位

```

```

return ngx_http_next_header_filter(r); //调用链表指针继续其他过滤模块的处理
}

```

//没有 if_modified_since 和 if_none_match 头，不需要其他操作，直接走后续链表处理
 return ngx_http_next_header_filter(r);

8.9 C++封装

ngx_http_request_t 结构体包含了太多的信息，是一个“万能类”，HTTP 的所有操作都与它相关，如果把它按照具体的功能分解成数个功能明确的小类就可以很好地降低复杂度，让它更容易理解和使用。

在 HTTP 处理过程中，主要的操作是处理请求数据和响应数据，所以可以定义两个类：NgxRequest 和 NgxResponse，它们整合了请求头/请求体和响应头/响应体的处理。

8.9.1 NgxHeaders

ngx_http_headers_in_t 和 ngx_http_headers_out_t 结构体可以理解为一个链表加若干指针信息，基本的功能仍然是操作链表。

因为请求头和响应头结构非常相似，所以我们实现一个通用的模板类 NgxHeaders。

类定义

NgxHeaders 的定义如下：

```

template<typename T,                                //T = ngx_http_headers_in|out_t
        T ngx_http_request_t::* ptr>
class NgxHeaders final : public NgxWrapper<T>

```

```

{
public:
    typedef NgxWrapper<T>                                super_type;
    typedef typename super_type::wrapped_type            ngx_headers_type;

    typedef NgxList<ngx_table_elt_t>                      ngx_headers_list_type;
    typedef ngx_table_elt_t                               kv_type;

public:
    NgxHeaders(ngx_headers_type& h):                        //从头结构体构造
        super_type(h),                                    //代理头结构体
        m_headers(h.headers)                              //代理头信息链表
    {}

    NgxHeaders(ngx_http_request_t* r):                     //从请求结构体构造
        NgxHeaders(r->*ptr)                               //使用成员变量指针获取头
    {}

    ~NgxHeaders() = default;

public:
    const ngx_headers_list_type& list() const              //访问头信息链表
    {
        return m_headers;
    }
    ...                                                    //其他成员函数，见后
private:
    ngx_headers_list_type m_headers;
};

```

NgxHeaders 代理了 ngx_http_headers_in_t/ngx_http_headers_out_t 对象，它还有一个 NgxList<ngx_table_elt_t> 成员变量 m_headers，用来操作头信息链表。

NgxHeaders 使用了与 4.3 节一样的成员变量指针技术，利用指针来获取 ngx_http_request_t 里的头结构体，所以请求头和响应头的类型定义就是：

```

typedef NgxHeaders<
    ngx_http_headers_in_t, &ngx_http_request_t::headers_in>
    NgxHeadersIn;                                         //请求头类型定义

typedef NgxHeaders<
    ngx_http_headers_out_t, &ngx_http_request_t::headers_out>
    NgxHeadersOut;                                       //响应头类型定义

```

查找头信息

非标准头不能直接用 ngx_http_headers_in_t 或 ngx_http_headers_out_t 里的快捷指针访问，我们必须遍历头信息链表，所以需要定义一个查找函数：

```

public:
    ngx_headers_list_type::iterator find(string_ref_type key) const
    {
        auto v = [&](const kv_type& kv) //查找谓词 lambda 表达式
        {
            return boost::iequals( //boost 大小写无关比较
                NgxString(kv.key).str(), key);
        };

        return list().find(v); //传入谓词进行查找
    }

```

find() 函数利用了 NgxList 的通用 find() 函数, 使用 boost::iequals 算法进行大小写无关比较以找到字符串对应的头信息。^①

访问头信息

find() 函数返回的是链表的迭代器, 使用迭代器就可以操作头信息:

```

public:
    bool has(string_ref_type key) const //是否存在头
    {
        return find(key) != list().end(); //与逾尾迭代器比较
    }

    ngx_str_t operator[](string_ref_type key) const //快捷访问头信息
    {
        auto p = find(key); //查找头

        if(p == list().end()) //检查是否找到
        {
            return ngx_null_string; //未找到返回空字符串
        }

        return p->value; //找到返回实际头信息
    }

```

重载的 operator[] 提供了快捷访问头信息的方法, 可以像字典那样直接用名字找到对应的值。

修改头信息

对头信息链表进行元素操作也可以添加或者删除头:

① 这里我们也可以使用 Nginx 提供的大小写无关比较函数 ngx_strncasecmp(), 但代码会比较烦琐, 读者可以自己尝试一下。

```

public:
    void add(const kv_type& kv) const //信息需在外部创建好
    {
        auto& tmp = list().prepare(); //链表添加元素
        tmp = kv;
    }

    void remove(string_ref_type key) const //“删除”头
    {
        auto p = find(key); //查找待删除的头

        if(p == list().end()) //未找到则无操作
        {
            return;
        }

        p->hash = 0; //hash 值置为 0
    }

```

ngx_list_t 实际上是没有删除元素操作的，所以我们需要把头信息的 hash 值置 0，表示无效，对于 Nginx 来说效果就相当于删除。

8.9.2 NgxRequestBody

类 NgxRequestBody 封装了对 Nginx 请求体的操作。

类定义

NgxRequestBody 的定义如下：

```

class NgxRequestBody final : public NgxWrapper<ngx_http_request_t>
{
public:
    typedef NgxWrapper<ngx_http_request_t> super_type;
    typedef NgxRequestBody this_type;

public:
    NgxRequestBody(ngx_http_request_t* r):super_type(r)
    {}
    ~NgxRequestBody() = default;

public:
    ... //其他成员函数，见后
};

```

因为对请求体的操作需要使用 ngx_http_request_t* 指针，所以 NgxRequestBody 使用 NgxWrapper 代理了 ngx_http_request_t 而不是 ngx_http_request_body_t。

丢弃或读取请求体

成员函数 `discard()` 和 `read()` 简单地封装了 Nginx 的 C 接口：

```
public:
    void discard() const                //丢弃请求体数据
    {
        auto rc = ngx_http_discard_request_body(get());

        NgxException::require(rc);      //检查错误码
    }

    template<typename F>                //要求传入回调函数
    ngx_int_t read(F f) const          //读取请求体数据
    {
        auto rc = ngx_http_read_client_request_body(get(), f);

        NgxException::fail(rc >= NGX_HTTP_SPECIAL_RESPONSE, rc);

        return NGX_DONE;               //处理尚未完成
    }
```

注意在读取请求体数据时我们必须返回 `NGX_DONE`，告诉 Nginx 处理尚未完成，这样才能在引擎里再次进入我们的模块，执行后续操作。

访问请求体数据

`bufs()` 检查请求里的 `ngx_http_request_body_t` 指针，返回缓冲区链表指针，使用它即可操作请求体数据：

```
public:
    ngx_chain_t* bufs() const           //获取请求体数据
    {
        return get()->request_body?    //是否有请求体
            get()->request_body->bufs : nullptr; //返回数据或空指针
    }
```

8.9.3 NgxRequest

类 `NgxRequest` 代理 `ngx_http_request_t` 的请求相关功能，集成了 `NgxHeadersIn` 和 `NgxRequestBody`。

类定义

`NgxRequest` 把 `NgxHeadersIn` 和 `NgxRequestBody` 作为自己的成员，定义如下：

```

class NgxRequest final : public NgxWrapper<ngx_http_request_t>
{
public:
    typedef NgxWrapper<ngx_http_request_t>    super_type;
    typedef NgxRequest                        this_type;

    typedef NgxHeadersIn                      headers_type;        //请求头类型
    typedef NgxRequestBody                    body_type;           //请求体类型
public:
    NgxRequest(ngx_http_request_t* r):super_type(r), //构造,代理 request_t
        m_headers(r), m_body(r)                    //初始化请求头和请求体
    {}
    ~NgxRequest() = default;
public:
    const headers_type& headers() const                //获取请求头
    {
        return m_headers;
    }

    const body_type& body() const                      //获取请求体
    {
        return m_body;
    }
    ...                                                //其他成员函数, 见后
private:
    headers_type    m_headers;                        //请求头
    body_type       m_body;                          //请求体
};

```

操作函数

对请求数据的处理大都可以通过调用 `headers()` 和 `body()` 的成员函数来完成, 但为了方便操作, `NgxRequest` 还提供了检查请求方法的功能:

```

public:
    //参数可以是 NGX_HTTP_GET|NGX_HTTP_HEAD|...
    bool method(ngx_uint_t x) const                //检查请求方法
    {
        return get()->method & x;
    }

```

读者也可以根据自己的需要为它添加其他常用的功能函数。

8.9.4 NgxResponse

NgxResponse 代理 ngx_http_request_t 里的响应相关功能, 因为没有响应体对应的数据结构, 所以它的代码略多一些。

类定义

NgxResponse 的定义如下:

```
class NgxResponse final : public NgxWrapper<ngx_http_request_t>
{
public:
    typedef NgxWrapper<ngx_http_request_t>    super_type;
    typedef NgxResponse                        this_type;

    typedef NgxHeadersOut                      headers_type;           //响应头类型
    typedef boost::string_ref                  string_ref_type;

public:
    NgxResponse(ngx_http_request_t* r):super_type(r),                //构造,代理 request_t
        m_headers(r), m_pool(r)                                       //初始化响应头和内存池
    {}

    ~NgxResponse() = default;

public:
    const headers_type& headers() const                             //获取响应头
    {
        return m_headers;
    }

    ...                                                                //其他成员函数, 见后

private:
    headers_type    m_headers;                                       //响应头
    NgxPool         m_pool;                                         //内存池对象
};
```

除响应头外, NgxResponse 还有一个内存池成员, 这是为了方便在发送数据时分配内存。

基本操作

NgxResponse 提取出了发送响应数据时最常用的设置状态和长度操作:

```
public:
    void status(ngx_uint_t x) const                                //设置状态码
    {
        headers()->status = x;
    }
```



```
void length(off_t len) const //设置数据长度
{
    headers()->content_length_n = len;
}
```

发送响应头

发送响应头时首先要检查 `header_sent` 标志位, 如果已经发送就不需要再继续操作。调用 `ngx_http_send_header()` 可以发送响应头, 如果返回的错误码不是 `NGX_OK`, 那么就意味着发生了错误, 可以用异常传递出去, 报告给 Nginx 框架处理:

```
public:
    ngx_int_t send() const //发送响应头
    {
        if(get()->header_sent) //检查发送标志位
        {
            return NGX_OK;
        }

        if(!headers()->status) //状态码是否已经设置
        {
            headers()->status = NGX_HTTP_OK; //默认值是 200
        }

        auto rc = ngx_http_send_header(get()); //发送响应头

        NgxException::fail(
            rc == NGX_ERROR || rc > NGX_OK, rc); //检查错误码

        return rc;
    }
```

发送响应体

基本的发送响应体操作是调用 `ngx_http_output_filter()` 发送 `ngx_chain_t` 数据。为了简化操作, 也是为了保证头已经正确发送, 可以先调用发送响应头操作:

```
public:
    ngx_int_t send(ngx_chain_t* out) const //发送响应体数据
    {
        send(); //确保已经发送响应头

        return out && !get()->header_only ? //检查是否有数据
            ngx_http_output_filter(get(), out) : NGX_OK; //发送响应数据
    }
```

我们还可以重载 `send()` 函数，支持直接发送 `ngx_buf_t`、`ngx_str_t` 和普通字符串：

```
ngx_int_t send(ngx_buf_t* buf) const                //发送 ngx_buf_t 对象
{
    NgxChainNode ch = m_pool.chain();                //创建 ngx_chain_t
    ch.set(buf);                                     //添加缓冲区到链里

    return send(ch);                                 //发送数据
}

ngx_int_t send(ngx_str_t* str) const                 //发送 ngx_str_t 对象
{
    NgxBuf buf = m_pool.buffer();                   //创建缓冲区对象
    buf.range(str);                                  //设置缓冲区数据

    return send(buf);                                //发送数据
}

ngx_int_t send(string_ref_type str) const            //发送 C 字符串
{
    auto s = m_pool.dup(str);                      //在内存池里复制一份
    return send(&s);                                 //发送数据
}
```

在最后一个 `send()` 函数里我们调用了内存池的 `dup` 操作，为字符串制作了一份备份，保证数据在发送时始终可用。

特殊操作

`NgxResponse` 的特殊操作有 `flush()`、`eof()` 和 `finalize()`：

```
public:
    ngx_int_t flush() const                          //刷新缓冲区，强制发送
    {
        return ngx_http_send_special(get(), NGX_HTTP_FLUSH);
    }

    ngx_int_t eof() const                             //结束数据的发送
    {
        return ngx_http_send_special(get(), NGX_HTTP_LAST);
    }

    void finalize(ngx_int_t rc = NGX_HTTP_OK) const    //结束当前请求的处理
    {
        ngx_http_finalize_request(get(), rc);
    }
```

```
}
```

8.10 开发 handler 模块

现在我们已经完全掌握了 Nginx 的 HTTP 框架和处理请求相关的知识，本节将使用现代 C++ 开发一个实用的 handler 模块。

8.10.1 模块设计

在模块开发之前，仍然要进行简单的设计：

- 模块名是 `ndg_echo_module`，是一个内容处理模块；
- 模块使用 `content handler` 的方式注册处理函数；
- 模块的功能是向客户端输出一个指定的字符串信息，由配置指令 `ndg_echo` 确定；
- `uri` 里的参数信息 (`$args`) 也一并输出；
- 配置指令是 `ndg_echo`，接受一个字符串参数。

8.10.2 配置信息类

类 `NdgEchoConf` 定义了模块所需的配置数据，根据设计，只有一个字符串成员：

```
class NdgEchoConf final
{
public:
    typedef NdgEchoConf this_type;           //简化类型定义
public:
    ngx_str_t msg;                           //存储配置文件里的字符串
    ...                                       //其他成员函数，见后
};
```

成员函数 `create()` 供 Nginx 框架调用，生成配置结构体：

```
public:
    static void* create(ngx_conf_t* cf)
    {
        return NgxPool(cf).alloc<this_type>(); //创建结构体对象
    }
```

我们还需要使用宏 `NGX_MOD_INSTANCE` 定义模块的单件类，方便其他代码来获取配置数据和环境数据：

```
NGX_MOD_INSTANCE(NdgEchoModule, ndg_echo_module, NdgEchoConf)
```

8.10.3 业务逻辑类

类 `NdgEchoHandler` 实现模块的业务逻辑，向客户端发送字符串：

```
class NdgEchoHandler final
{
    typedef NdgEchoModule  this_module;           //简化类型定义
public:
    static ngx_int_t handler(ngx_http_request_t *r) //处理函数
    try                                           //try-catch 捕获异常
    {
        NgxRequest req(r);                     //请求对象

        if(!req.method(NGX_HTTP_GET))           //检查请求方法
        {
            return NGX_HTTP_NOT_ALLOWED;         //要求必须是 get
        }

        req.body().discard();                   //丢弃请求体

        auto& cf = this_module::conf().loc(r);   //获取配置数据
        NgxString msg = cf.msg;                  //获取配置里的字符串

        NgxString args = req->args;              //获取 uri 的参数

        auto len = msg.size();                   //计算响应体的长度
        if(!args.empty())                        //需要加上$args 的长度
        {
            len += args.size()+1;                //还有一个“,”的长度
        }

        NgxResponse resp(r);                     //准备发送响应

        resp.length(len);                         //设置响应体长度
        resp.status(NGX_HTTP_OK);                 //设置状态码

        if(!args.empty())                        //如果有$args
        {
            resp.send(args);                      //原样发送$args
            resp.send(",");                       //再发送一个逗号
        }

        resp.send(msg);                           //发送配置字符串

        return resp.eof();                       //eof 表示发送结束
    }
};
```

```

    }
    catch(const NgxException& e)                //try-catch 捕获异常
    {
        return e.code();                        //返回错误码
    }
};                                              //业务逻辑类结束

```

NdgEchoHandler 类里只有一个 handler() 函数用来处理请求，它可以分为两个部分。前半段使用 ngx_request 检查请求头参数，用 NdgEchoModule 获取配置数据，准备响应数据。后半段使用 ngx_response 设置了最基本的状态码和响应体长度，然后调用 send() 发送数据。

我们并没有直接调用 send() 来发送响应头，这是因为发送响应头的操作已经隐含在了发送响应体的操作里。

最后我们必须调用 eof() 函数，它实际上是发送了一个 last_buf 标志位为 1 的 ngx_buf_t 对象，表示 HTTP 请求处理结束。

8.10.4 模块集成类

类 NdgEchoInit 整合配置信息类和业务逻辑类，实现模块集成。

类定义

NdgEchoInit 的定义如下：

```

class NdgEchoInit final
{
public:
    typedef NdgEchoConf      conf_type;          //简化类型定义
    typedef NdgEchoHandler   handler_type;
    typedef NdgEchoInit      this_type;

public:
    ...
};
//其他成员函数，见后

```

配置指令解析

因为模块使用了 content handler 的注册方式，所以我们必须实现自己的指令解析函数，在函数里向 ngx_http_core_module 注册处理函数：

```

private:
    static char* set_echo(ngx_conf_t* cf, ngx_command_t* cmd, void* conf)
    {

```

```

auto rc =                                     //调用 Nginx 的解析函数
    ngx_conf_set_str_slot(cf, cmd, conf);

if(rc != NGX_CONF_OK)                        //检查是否解析成功
{
    return rc;
}

NgxHttpCoreModule::handler(                //设置 location 处理函数
    cf, &handler_type::handler);

return NGX_CONF_OK;
}

```

随后我们定义 `cmds()` 函数，设置指令数组：

```

public:
    static ngx_command_t* cmds()
    {
        static ngx_command_t n[] =           //配置指令数组，静态变量
        {
            {
                ngx_string("ndg_echo"),
                NgxTake(NGX_HTTP_LOC_CONF, 1), //只使用一个参数
                &this_type::set_echo,           //自定义指令解析函数
                NGX_HTTP_LOC_CONF_OFFSET,      //只能在 location 里出现
                offsetof(conf_type, msg),      //解析使用的字段偏移量
                nullptr
            },

            ngx_null_command                  //空对象，结束数组
        };

        return n;                           //返回数组地址
    }

```

函数指针表

`ndg_echo_module` 是 content handler 模块，不需要 postconfiguration 执行初始化操作，也没有合并操作，它只工作在 location 层次，这与第6章的 `ndg_test_module` 不同：

```

public:
    static ngx_http_module_t* ctx()
    {
        static ngx_http_module_t c =
        {

```

```

    NGX_MODULE_NULL(6),
    &conf_type::create,           //创建 location 域的配置结构
    NGX_MODULE_NULL(1),
};

return &c;
}

```

模块定义

最后还要使用 `ctx()` 和 `cmds()` 来定义 `ngx_module_t` 对象:

```

public:
    static const ngx_module_t& module()
    {
        static ngx_module_t m =           //模块定义, 静态变量
        {
            NGX_MODULE_V1,                //标准的填充宏
            ctx(),                        //配置功能函数
            cmds(),                       //配置指令数组
            NGX_HTTP_MODULE,              //http 模块必须的 tag
            NGX_MODULE_NULL(7),           //不使用的 7 个函数指针
            NGX_MODULE_V1_PADDING        //标准的填充宏
        };

        return m;                        //返回对象的常量引用
    }

```

8.10.5 实现源文件

在 `cpp` 文件里我们定义 Nginx 编译所需的 `ngx_module_t` 变量:

```

#include "NdgEchoInit.hpp"           //包含头文件
auto ndg_echo_module = NdgEchoInit::module(); //定义 ngx_module_t 变量

```

8.10.6 编译脚本

`ndg_echo_module` 使用的编译脚本也很简单, 只需设置四个 Shell 变量, 最后调用“`auto/module`”即可:

```

ngx_addon_name=ndg_echo_module      #模块名字
ngx_module_type=HTTP                 #模块的类型, http handler 模块
ngx_module_name=ndg_echo_module      #模块的编译用名字
ngx_module_srcs="$ngx_addon_dir/ModNdgEcho.cpp" #模块的源码

```

`. auto/module` #调用 Nginx 的模块脚本

执行“`./configure`”脚本把模块添加进 Nginx 框架再 `make` 就完成了模块的开发:

`./configure --add-module=path/to/echo;make` #静态模块集成

8.10.7 测试验证

在 Nginx 的配置文件里定义如下的 location, 加入 `ndg_echo` 指令:

```
location /echo {
    ndg_echo "hello nginx\n";          #启用 ndg_echo_module 模块
}
```

使用 `curl` 就可以测试我们刚刚开发完成的新模块:

```
curl http://localhost/echo          #输出 hello nginx
curl http://localhost/echo?chrono   #输出 chrono,hello nginx
```

8.11 开发 filter 模块

本节将使用第 7 章实现的 `NgxFooter` 类开发一个 filter 模块, 它同时使用了 `header filter` 和 `body filter`, 并且展示了请求的环境数据结构 (`ctx`) 的用法。

8.11.1 模块设计

对这个 filter 模块的基本设计如下:

- 模块名是 `ndg_footer_module`, 过滤处理响应头和响应体;
- 配置指令 `ndg_header` 可以接受多个 `keyval` 参数, 加入到响应头;
- 配置指令 `ndg_footer` 接受一个字符串参数, 加入到响应体末尾;
- 两个指令不需要同时出现, 也就是说可以只修改响应头或者只修改响应体;
- 需要使用 `ctx` 来记录状态, 防止重复添加。

8.11.2 配置信息类

类 `NdgFooterConf` 定义了模块所需的配置数据, 它有两个成员, 分别存储响应头和响应体的修改信息:

```
class NdgFooterConf final
{
public:
```



```

typedef NdgFooterConf this_type;           //简化类型定义
public:
    ngx_array_t*    headers;               //存储修改响应头的 kv 数据
    ngx_str_t        footer;               //存储修改响应体的字符串
    ...                                     //其他成员函数，见后
};

```

注意在 Nginx 里存储 ngx_keyval_t 数据时必须使用 ngx_array_t* 的形式，这样才能够被解析函数 ngx_conf_set_keyval_slot() 正确处理。

成员函数 create() 生成配置结构体：

```

public:
    static void* create(ngx_conf_t* cf)
    {
        return NgxPool(cf).alloc<this_type>();
    }

```

8.11.3 环境数据类

因为响应头和响应体的过滤处理是彼此独立的两个过程，所以在开发 filter 模块时通常都需要使用环境数据对象 ctx，在里面放置一些共享的信息，用来保证过滤处理的一致性。

一种比较常用的方式是直接用 ngx_http_get_module_ctx() 检查模块对应的 ctx 是否存在，如果不存在那么 header filter 就创建 ctx，而 body filter 就不需要处理。

ndg_footer_module 的功能比较简单，所以 ctx 里只有一个整数标志变量，记录当前处理的阶段，具体含义是：

- 0 : 还未开始处理，需要处理响应头；
- 1 : 响应头已经处理，需要处理响应体；
- 2 : 响应体已经处理完毕。

环境数据类 NdgFooterCtx 定义如下：

```

struct NdgFooterCtx final
{
    int flag = 0;           //过滤处理的标志位
};

```

宏 NGX_MOD_CTX_INSTANCE 使用配置信息类和环境数据类定义模块的单件：

```

NGX_MOD_CTX_INSTANCE(NdgFooterModule,
    ndg_footer_module, NdgFooterCtx, NdgFooterConf)

```

8.11.4 业务逻辑类

类 `NdgFooterHandler` 使用 `NgxFilter` 向 Nginx 过滤链表注册自己的处理函数，并实现对响应数据的过滤处理。

在编写过滤函数时我们不能简单地返回 `NGX_OK` 这样的错误码，必须要使用 `next` 指针调用返回，否则会导致过滤链表中断，无法正确发送数据。

比较好的做法是使用模板方法模式，用一个辅助函数执行处理逻辑，而在主函数里调用后续的链表操作，这样就保证了过滤链表能够正确执行。

类定义

`NdgFooterHandler` 的定义如下，注意 `NgxFilter` 的用法：

```
class NdgFooterHandler final
{
public:
    typedef NdgFooterHandler      this_type;      //简化类型定义
    typedef NdgFooterModule      this_module;
    typedef NgxFilter<this_type> this_filter;      //过滤器工具类
    ...                                           //其他成员函数，见后
};
```

代码里最重要的是模板类 `NgxFilter` 的实例化，它使用 `NdgFooterHandler` 作为编译期 tag，实例化出了专用的过滤器指针。

模块初始化

`init()` 函数用来初始化 `filter` 模块，会作为 `postconfiguration` 函数指针被 Nginx 调用，把处理函数插入过滤链表。`NgxFilter` 已经封装了过滤链表的头节点操作，所以我们的代码很简单：

```
public:
    static ngx_int_t init(ngx_conf_t* cf)          //postconfiguration 调用
    {
        this_filter::init(                          //插入过滤链表
            &this_type::header_filter, &this_type::body_filter);
        return NGX_OK;
    }
```

处理响应头

通常情况下响应头过滤函数只会被调用一次，所以可以在这里执行模块的初始化操作，设

置环境数据。

`header_filter()` 函数调用辅助函数, 然后再执行后续的过滤链表(省略了 `try-catch` 捕获异常的代码):

```
static ngx_int_t header_filter(ngx_http_request_t *r)
{
    do_header_filter(r);                //辅助函数, 真正的业务逻辑
    return this_filter::next(r);        //保证过滤链表可以继续处理
}
```

辅助函数 `do_header_filter()` 的基本工作流程是: ①

- 1) 检查 `ctx`, 如果不是 0, 则表示处理阶段不对, 调用后续链表;
- 2) 设置 `flag` 为 1;
- 3) 获取配置数据 `headers`, 遍历存储的 `kv` 值, 加入到响应头;
- 4) 获取配置数据 `footer`, 如果有则修改 `content_length`;

`do_header_filter()` 的实现代码如下:

```
static void do_header_filter(ngx_http_request_t *r)
{
    auto& ctx = this_module::data(r);    //获取环境数据
    if(ctx.flag)                          //要求必须是 0, 表示还未处理
    { return; }                           //退出, 继续链表的后续处理

    ctx.flag = 1;                         //置标志为 1
    ngx_response resp(r);                 //准备处理响应头

    auto& cf = this_module::conf().loc(r); //获取配置数据

    ngx_kv_array headers = cf.headers;    //获取要增加的头

    for(auto i = 0u; i < headers.size(); ++i) //遍历数组
    {
        auto& header = headers[i];        //取数组元素

        ngx_table_elt_t kv;
        kv.hash = 1;                      //置 hash 为 1, 表示有效
        kv.key = header.key;               //头的 key
    }
```

① 为了简化代码, `filter` 的处理逻辑省略了 `r->header_only`、`r->method` 等请求属性的检查。

```

kv.value = header.value; //头的 value

resp.headers().add(kv); //加入响应头里
}

NgxString footer = cf.footer; //获取要添加的字符串
if(footer.empty()) //空字符串则不添加
{ return; }

auto len = resp.headers()->content_length_n; //获取原响应体长度
if(len > 0) //已有长度
{
    resp.length(len + footer.size()); //重新设置长度
}
//函数结束，继续链表的后续处理

```

处理响应体

body_filter() 函数使用同样的模板方法模式处理响应体：

```

static ngx_int_t body_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    do_body_filter(r, in); //辅助函数，真正的业务逻辑
    return this_filter::next(r, in); //继续链表的后续处理
}

```

因为过滤链表的执行是在 ngx_http_output_filter() 里，已经是在数据的发送过程中，所以我们不能也没有必要再调用这个函数，处理响应数据主要就是操作 ngx_buf_t 和 ngx_chain_t 数据结构，添加数据，调整链表：

```

static void do_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    auto& cf = this_module::conf().loc(r); //获取配置数据

    NgxString footer = cf.footer; //获取要添加的字符串
    if(footer.empty()) //空字符串则不添加
    { return; } //继续链表的后续处理

    auto& ctx = this_module::data(r); //获取环境数据

    if(ctx.flag != 1) //要求必须是1，即响应头已经处理，响应体还未处理
    { return; } //继续链表的后续处理

    NgxChain ch = in; //检查响应数据链
    auto p = ch.begin(); //迭代器初始化
}

```

```

for(; p != ch.end();++p) //迭代器遍历链表
{
    if(p->data().last()) //查找数据末尾, 即 eof
    { break; }
}

if(p == ch.end()) //未找到 eof
{ return; } //继续链表的后续处理

ctx.flag = 2; //置标志为 2, 开始处理

NgxPool pool(r); //内存池准备分配数据空间

NgxBuf buf = pool.buffer(); //创建一块缓冲区
buf.range/footer); //添加字符串
buf.finish(); //设置为 eof

if(!p->data().size()) //检查原链表的数据是否为空
{
    p->set(buf); //为空则直接设置
    return; //继续链表的后续处理
}

NgxChainNode n = pool.chain(); //创建一个链表节点
n.set(buf); //缓冲区添加到链表
n.finish(); //链表结束

p->link(n); //链接到原链表
p->data().finish(false); //修改原链表的 eof 标志
} //函数结束, 继续链表的后续处理

```

8.11.5 模块集成类

filter 模块的集成类 `NdgFooterInit` 与 `handler` 模块类似, 故仅列出部分主要代码:

```

static ngx_command_t* cmds() //指令数组函数
{
    static ngx_command_t n[] = //配置指令数组, 静态变量
    {
        {
            ngx_string("ndg_header"), //添加头信息的指令
            NgxTake(NGX_HTTP_LOC_CONF, 2),
            ngx_conf_set_keyval_slot, //Nginx 的解析 kv 函数
            NGX_HTTP_LOC_CONF_OFFSET,
            offsetof(conf_type, headers),
        }
    }
}

```

```

        nullptr
    },

    {
        ngx_string("ndg_footer"),           //添加末尾字符串指令
        NgxTake(NGX_HTTP_LOC_CONF, 1),
        ngx_conf_set_str_slot,              //Nginx 的解析函数
        NGX_HTTP_LOC_CONF_OFFSET,
        offsetof(conf_type, footer),
        nullptr
    },

    ngx_null_command
};

return n;
}

static ngx_http_module_t* ctx()           //函数指针表
{
    static ngx_http_module_t c =
    {
        NGX_MODULE_NULL(1),
        &handler_type::init,              //初始化过滤链表
        NGX_MODULE_NULL(4),
        &conf_type::create,
        NGX_MODULE_NULL(1),
    };

    return &c;
}

```

这段代码里最重要的是设置模块的 `postconfiguration` 函数指针，在解析配置文件后初始化过滤链表，把处理函数添加到链表的正确位置。

8.11.6 实现源文件

cpp 文件中定义 Nginx 编译所需的 `ngx_module_t` 变量，只需要两行代码：

```

#include "NdgFooterInit.hpp"
auto ndg_footer_module = NdgFooterInit::module();

```

8.11.7 编译脚本

`ndg_footer_module` 使用的编译脚本同样很简单，但需要注意使用的变量 `ngx_module_type` 的值应该是“`HTTP_FILTER`”，千万不要误用：

```
ngx_addon_name=ndg_footer_module           #模块的名字
ngx_module_type=HTTP_FILTER                #注意类型是 HTTP_FILTER
ngx_module_name=ndg_footer_module           #模块编译用的名字
ngx_module_srcs="$ngx_addon_dir/ModNdgFooter.cpp" #模块的源码

. auto/module                               #调用 Nginx 的模块脚本
```

8.11.8 测试验证

我们可以利用 8.10 节定义的 location “`/echo`”，过滤处理 `ndg_echo` 指令：

```
location /echo {
    ndg_echo "hello nginx\n";                #启用 ndg_echo_module 模块

    ndg_header x-name chrono;                 #启用添加头过滤处理
    ndg_header x-value trigger;              #可以添加多个头信息

    ndg_footer "ocarina of time\n";          #启用添加末尾字符串过滤处理
}
```

用 `curl` 测试 `ndg_footer_filter` 模块：

```
curl -v http://localhost/echo?zelda
```

返回的数据是：

```
< HTTP/1.1 200 OK
< Server: nginx/1.12.0
< Content-Length: 34
< x-name: chrono                    #新添加的头信息
< x-value: trigger
zelda,hello nginx
ocarina of time                     #新添加的末尾字符串
```

8.12 总结

本章主要研究了 Nginx 处理 HTTP 请求的核心结构体 `ngx_http_request_t`。它是 HTTP 框架里最重要的数据结构之一，可以说是 Nginx 宫殿里的“镇宅之宝”，包含了处理过

程中需要的所有信息，本章及之后的章节处理 HTTP 请求时都需要操作 `ngx_http_request_t` 中的各个成员。

依据 RFC2616，HTTP 协议由请求头、请求体、响应头和响应体等组成，`ngx_http_request_t` 使用 `headers_in`、`request_body` 和 `headers_out` 等成员与之对应。

在接收请求数据的同时 Nginx 解析了请求行和请求头，把请求方法、资源标识符、协议版本号和头的键值对等信息都存储到了 `ngx_http_request_t` 里，可以很容易地访问，也可以操作它们来改写请求的参数。响应头和响应体的操作的处理比较简单，只要设置好数据再调用函数发送就可以了，Nginx 框架会自动做好剩余的事情。

本章的后半部分内容以实例讲解了如何开发 handler 模块和 filter 模块。

开发 handler 模块最重要的工作是实现并注册 `ngx_http_handler_pt` 处理函数。在处理函数里，我们可以用 `NgxRequest` 来代理 `ngx_http_request_t` 对象，操作 HTTP 请求的各个字段，丢弃或者读取请求体。如果是 content handler 模块，就需要结合配置信息决定产生的响应数据，再用 `NgxResponse` 的 `send()` 函数发送响应头和响应体，从而完成请求的处理。

在创建响应数据时切记不能在栈上分配内存（函数内部的局部变量），因为 Nginx 在后续的异步处理时很可能栈内存已经无效，会发生内存越界错误，应当尽量使用请求里的内存池来分配内存。

向 Nginx 框架注册处理函数可以使用 `NgxHttpCoreModule` 类，它很好地封装了 Nginx 的两种 handler 注册机制，只需要一行代码就可以轻松完成注册工作。

filter 模块的开发与 handler 模块略有不同，它主要的处理对象是响应头和 `ngx_chain_t` 里的数据块，业务逻辑较为简单。因为过滤操作是异步的，而且响应数据可能有多份，所以 filter 模块通常需要使用 `ctx` 来记录处理状态，并在 header filter 里初始化 `ctx`。

实现自己的过滤处理函数通常需要使用 `next` 指针调用后续的过滤链表，否则会导致 Nginx 无法正确发送数据，使用模板方法模式可以很好地避免这个问题。

`NgxFilter` 类封装了 Nginx 过滤链表的基本操作，包括初始化和调用后续链表，简化了 filter 模块的开发工作。

第 9 章

Nginx HTTP请求转发

handler 模块和 filter 模块是 Nginx 里数量最多的模块，提供了非常丰富的功能，但它们也有“致命”的“弱点”——只能处理本地的资源，能力被限制在了单服务器范围内。如果仅有这两类模块，无疑会使 Nginx 的吸引力大打折扣。

为了访问外部资源，Nginx 在 HTTP 处理框架里又实现了 upstream 框架，提供了全异步、高性能的请求转发机制，能够让 Nginx 超越单机的限制，访问任意的后端应用服务器收发数据，获得“他山之石可以攻玉”的扩展能力。

upstream 框架也是 Nginx 反向代理的基础，能够让 Nginx 成为网络里的核心节点，构建起大型的服务网络。

9.1 框架简介

ngx_http_upstream_module 是 upstream 框架的实现模块，它在 Nginx 的 HTTP 框架里定义了无阻塞访问后端服务器的机制，使用负载均衡算法选择后端，再用回调函数来处理后端返回的数据，不仅可以支持 HTTP 协议，还可以支持 FastCGI、Memcached、Redis 等任意的协议，非常灵活。

Nginx 选择了术语“upstream”而不是常见的“backend”，这是因为在复杂的网络中 backend 一词不够准确，一个所谓的 backend 可能并不是后端服务器而是前端服务器，含义容易混淆，而 upstream 能够更好更形象地描述 Nginx 在网络数据流里承上启下的位置和作用。

9.1.1 工作原理

`ngx_http_upstream_module` 是一个比较特殊的 http 模块，它相当于 `content handler` 模块，工作在 `NGX_HTTP_CONTENT_PHASE`，向上游服务器发起 TCP 连接，转发请求，获取响应数据，然后返回 HTTP 数据给下游客户端。

但 `ngx_http_upstream_module` 又是一个“不完整”的模块，不能独立处理请求，而是需要其他模块配合才能工作。它实现了大部分的底层网络收发逻辑和 Nginx 处理流程，并且在一些关键点定义了回调函数，其他模块需要实现这些函数，然后“插入”进流程来实现负载均衡算法或者处理特定上游服务器的数据。

为了叙述方便，本章之后称 `ngx_http_upstream_module` 为 upstream 框架，而实现回调函数的模块称为 upstream 模块和 load-balance 模块。

在整个请求转发的过程中 Nginx 扮演着“中间人”的角色，把下游的请求转发给上游的服务器（这里不一定是 HTTP 协议），对接收到的数据做适当的处理后再返回给下游。从下游的客户端来看，Nginx 的请求转发处理是完全透明的。

Nginx 请求转发机制的基本工作流程如下：

- 1) upstream 框架初始化 load-balance 模块；
- 2) upstream 模块在 HTTP 框架里注册处理函数，准备处理客户端请求；
- 3) upstream 模块设置 upstream 的一些连接参数，如超时时间、缓冲区大小；
- 4) upstream 模块调用 `ngx_http_upstream_init()` 启动 upstream 机制；
- 5) upstream 框架回调 `create_request()`，得到要发送的请求；
- 6) upstream 框架调用 load-balance 模块，选择一个上游服务器地址；
- 7) upstream 框架连接上游服务器，异步与上游服务器交互，收发数据；
- 8) upstream 框架回调 `process_header()`，处理收到的响应头；
- 9) upstream 框架回调 `input_filter_init()` 和 `input_filter()` 处理响应体；
- 10) upstream 框架回调 `finalize_request()`，执行收尾工作；
- 11) HTTP 框架发送响应头和响应体数据。

可以看到，upstream 框架的工作流程与 HTTP 框架的工作流程有些相似，它一方面要接

收来自客户端的数据，另一方面要接收来自上游服务器的数据。前者由 HTTP 框架处理，而后者则由 upstream 框架处理，同样也要解析响应头和响应体，但这些数据不一定是 HTTP 协议，Nginx 无法控制，所以它把数据的解析处理工作交给了外部 upstream 模块的回调函数去实现，从而能够灵活支持 HTTP、FastCGI、Memcached、Redis 等协议。

upstream 框架的工作流程可以用图 9-1 来表示。

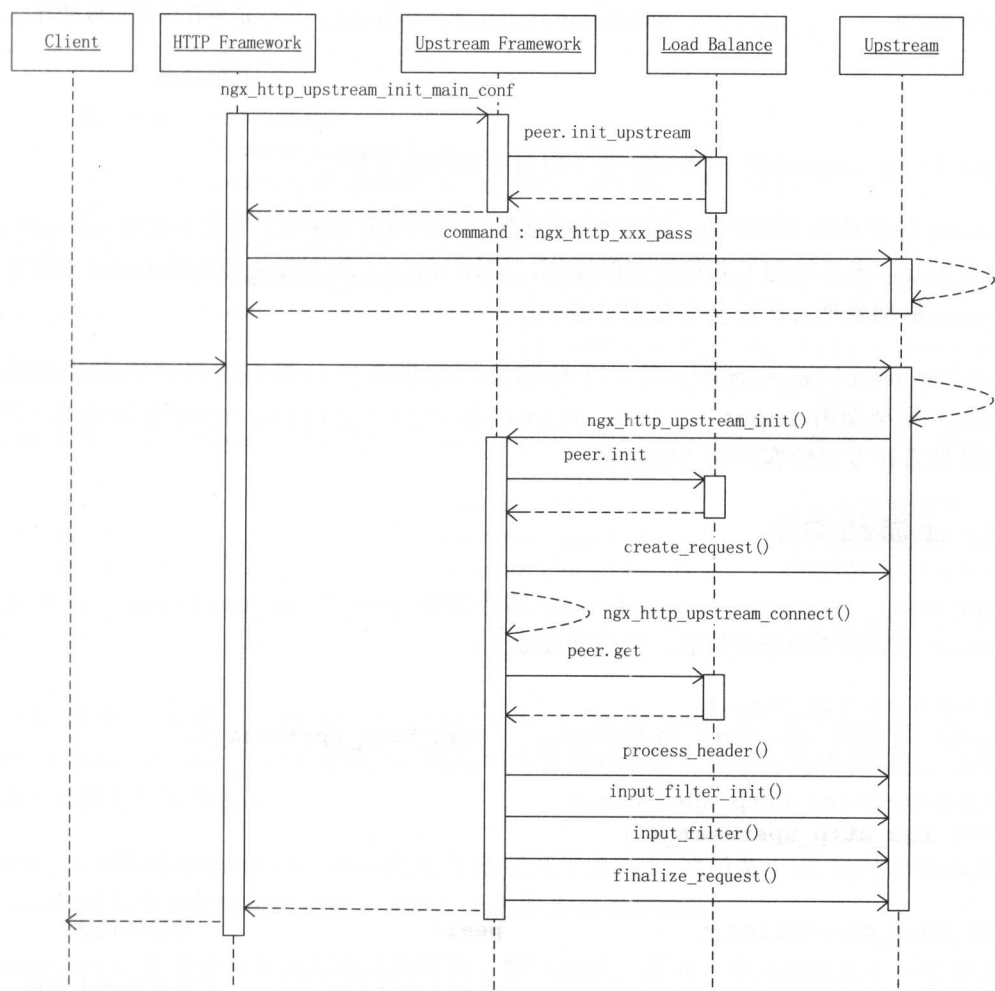


图 9-1 upstream 框架的工作流程

9.1.2 请求结构体

Nginx 的请求转发仍然离不开核心结构体 ngx_http_request_t，它里面的成员

upstream 是请求转发机制的关键:

```
// 定义在 http/nginx_http_request.h
struct ngx_http_request_s {
    ngx_http_upstream_t*    upstream;           //请求转发的关键结构体

    ngx_http_request_t*     main;               //主请求指针
    unsigned                 count:8;           //引用计数
    unsigned                 subrequest_in_memory:1; //是否在内存里做处理

    ...                                           //其他成员
};
```

ngx_http_request_t 里还有两个成员与请求转发有关。

count 是请求的引用计数, 是 Nginx 处理请求时的重要参数。当我们使用 upstream 时必须把它加 1, 表示发起了其他的关联操作, 这样 Nginx 就不会关闭连接销毁请求对象, 当 upstream 框架处理完毕后会自动把它减 1。

subrequest_in_memory 是一个标志位, 如果是 1, 那么收到的响应数据就会由 upstream 框架调用 input_filter_init() 和 input_filter() 进行过滤处理, 否则不做任何处理直接发送给客户端 (即透传)。

9.1.3 上游结构体

ngx_http_request_t::upstream 的类型是 ngx_http_upstream_t, 它定义了 upstream 机制需要的所有信息, 代码摘要如下:

```
//定义在 http/nginx_http.h
typedef struct ngx_http_upstream_s    ngx_http_upstream_t;

//定义在 http/nginx_http_upstream.h
struct ngx_http_upstream_s {
    ...                                           //其他成员

    ngx_peer_connection_t                    peer;           //连接结构体

    ngx_chain_t*                             request_bufs;    //发送的请求数据

    ngx_http_upstream_conf_t*                conf;           //上游的连接参数设置

    ngx_http_upstream_headers_in_t           headers_in;     //上游的响应头

    ngx_http_upstream_resolved_t*            resolved;       //上游服务器的地址
};
```

```

ngx_buf_t      buffer;           //数据缓冲区
off_t          length;          //缓冲数据的长度

ngx_chain_t*   out_bufs;        //从上游接收到的数据

//处理上游服务器响应数据的回调函数指针
ngx_int_t      (*input_filter_init)(void *data);
ngx_int_t      (*input_filter)(void *data, ssize_t bytes);
void           *input_filter_ctx;

//发送接收请求的回调函数指针
ngx_int_t      (*create_request)(ngx_http_request_t *r);
ngx_int_t      (*reinit_request)(ngx_http_request_t *r);
ngx_int_t      (*process_header)(ngx_http_request_t *r);
void           (*abort_request)(ngx_http_request_t *r);
void           (*finalize_request)(ngx_http_request_t *r, ngx_int_t rc);
ngx_int_t      (*rewrite_redirect)(ngx_http_request_t *r,
                                   ngx_table_elt_t *h, size_t prefix);
ngx_int_t      (*rewrite_cookie)(ngx_http_request_t *r, ngx_table_elt_t *h);

ngx_http_upstream_state_t  state;           //处理的状态信息

unsigned        buffering:1;               //是否使用缓冲
unsigned        request_sent:1;           //是否已经发送请求
unsigned        header_sent:1;            //是否已经发送响应头

...                                         //其他成员
};

```

从结构定义来看, `ngx_http_upstream_t` 很像 `ngx_http_request_t`, 也有 `headers_in`、`header_sent` 等成员, 这是因为它需要连接上游服务器收发数据, 而这些成员用来记录数据收发的状态。

`ngx_http_upstream_t` 特别的地方是定义了九个回调函数, 这些回调函数需要由 `upstream` 模块来实现特定的功能, 才能与上游服务器进行通信。

`upstream` 框架也对部分回调函数提供了默认处理, 所以九个回调函数不一定要全部实现, 只有四个函数是必须由 `upstream` 模块提供的, 它们是:

- `create_request` : 生成发送到上游的请求数据;
- `reinit_request` : 发生错误时重新初始化请求数据;
- `process_header` : 解析收到的响应头;

- `finalize_request` : 请求结束时的收尾工作。

9.1.4 上游配置参数

在请求转发过程中 Nginx 会作为一个客户端发送 TCP 请求, 所以需要设置超时时间、缓冲、限速等必要的参数, 才能正确地连接上游服务器。

Nginx 使用结构体 `ngx_http_upstream_conf_t` 来配置 upstream 框架连接上游服务器的参数, 定义摘要如下:

```
//定义在 http/ngx_http_upstream.h
typedef struct {
    ngx_http_upstream_srv_conf_t* upstream; //upstream 框架的 srv 配置信息

    ngx_msec_t          connect_timeout; //连接超时时间
    ngx_msec_t          send_timeout;   //发送超时时间
    ngx_msec_t          read_timeout;   //接收超时数据

    size_t              buffer_size;    //缓冲区大小

    ngx_bufs_t          bufs;           //缓冲区数量设置
    ngx_flag_t          buffering;      //是否使用接收缓冲区

    ... //其他成员
} ngx_http_upstream_conf_t;
```

`ngx_http_upstream_conf_t` 里可以设置的参数有很多, 但只有少数成员是我们目前需要关心的。

`xxx_timeout` 用来设置 upstream 框架连接上游服务器的超时时间, 它们的单位是毫秒, 可以在配置文件里使用 Nginx 提供的 `ngx_conf_set_msec_slot()` 函数来设置。

`buffer_size` 设置了 upstream 框架接收数据的缓冲区大小, 可以用 `ngx_conf_set_size_slot()` 函数来设置。我们也可以使用 Nginx 的全局变量 `ngx_pagesize`, 它实际上就是系统默认的页面大小:

```
ngx_pagesize = getpagesize(); //定义在 os/unix/ngx_posix_init.c
```

在 `ngx_http_upstream_conf_t` 结构体里也有一个 `upstream` 成员, 但它与 `ngx_http_request_t` 里的 `upstream` 完全不同, 它表示的是 Nginx 配置文件里的上游服务器地址配置信息, upstream 框架用它来连接上游服务器。

9.2 请求转发机制

upstream 框架工作在 Nginx 的 HTTP 框架之内，由 HTTP 处理引擎调度，在内容产生阶段从上游获取数据再转发给下游，所以它的主要任务就是与上下游的交互并处理数据。

从 upstream 模块的角度来看，使用 upstream 机制要有如下几个步骤：

- 1) 定义处理上游数据的回调函数；
- 2) 调用函数 `ngx_http_upstream_create()` 初始化 upstream 机制；
- 3) 设置 upstream 的连接参数，包括上游服务器的地址、超时时间等；
- 4) 调用 `ngx_http_upstream_init()` 启动 upstream 机制；
- 5) 使用 `create_request()` 提供请求数据；
- 6) 使用 `process_header()` 和 `input_filter()` 处理收到的数据；
- 7) 使用 `finalize_request()` 执行结束时的清理工作。

9.2.1 回调函数

`ngx_http_upstream_t` 结构体定义了 upstream 模块需要实现的九个回调函数，它们是：

- `create_request` : 构造待发送到上游的请求数据；
- `reinit_request` : 如果连接上游失败，则在重连前执行初始化；
- `process_header` : 解析收到的响应头数据；
- `abort_request` : 暂无意义，目前 upstream 框架未调用；
- `finalize_request` : 请求结束时的收尾工作；
- `rewrite_redirect` : 响应头重定向操作；
- `rewrite_cookie` : 改写 cookie 操作；
- `input_filter_init` : 过滤响应体前的初始化；
- `input_filter` : 过滤收到的响应体数据；

虽然回调函数比较多，但 upstream 模块可以根据自身的具体情况只实现部分函数，如果回调函数是空指针，upstream 框架要么跳过处理步骤，要么使用默认的函数来处理。

下面简要介绍几个实际开发中较为常用的回调函数。

create_request

`create_request` 在 `upstream` 机制启动时被调用, 生成能够与上游服务器正确通信的请求数据, 例如 HTTP 请求头、Memcached 命令等, 并把数据以 `ngx_chain_t` 的形式存放在 `ngx_http_upstream_t::request_bufs` 里, `upstream` 框架在连接上游成功后就会发送请求。

reinit_request

如果连接上游服务器失败, 或者连接成功但发送失败, 这时 `upstream` 结构体就处于一种不正确的状态, 必须要重新初始化才能连接下一个服务器。这时 `upstream` 框架会调用模块的 `reinit_request` 函数, 要求模块重新初始化请求相关的数据。

通常 `reinit_request` 的工作和 `create_request` 是基本相同的, 但由于两者运行的时机不同, 所以也可能会有不同的处理逻辑。

process_header

`upstream` 框架把从上游收到的响应数据存储在 `ngx_http_upstream_t::buffer` 里, `process_header` 的作用就是把数据分解出响应头和响应体两部分, 并把响应头里的状态码、内容长度等信息存储到 `headers_in` 和 `state` 成员, 转化为下游可以理解的 HTTP 格式, 这个过程与 HTTP 框架的 `ngx_http_process_request_line()` 很相似。

由于 `upstream` 框架异步接收数据, `process_header` 不可能一次就解析完毕, 所以通常需要使用请求环境数据 `ctx` 来暂存解析结果, 如果未解析完返回 `NGX_AGAIN` 继续解析, 如果返回 `NGX_OK` 则表示响应头解析完毕, 其后的数据都是响应体。

input_filter_init 和 input_filter

响应体数据也存放在 `buffer` 里, 当 `ngx_http_request_t` 里的 `subrequest_in_memory` 标志位是 1 时, `upstream` 框架就会调用 `input_filter_init` 和 `input_filter` 这两个函数对响应体执行过滤处理, 加工出对下游有效的数据。

如果这两个回调函数为空, 那么 `upstream` 框架会使用默认的 `ngx_http_upstream_non_buffered_filter_init()` 和 `ngx_http_upstream_non_buffered_filter()` 函数, 它们只是简单地把 `buffer` 里收到的数据依次挂到 `out_bufs` 链表的末尾, 这意味着 `upstream` 框架设置的 `buffer` 必须足够大, 能够容纳上游的所有数据。

finalize_request

当响应数据发送完毕或者发生错误时，upstream 框架会执行 `ngx_http_upstream_finalize_request()` 函数结束请求，这时会调用 `finalize_request` 回调函数，让 upstream 模块有机会做一些自己的收尾工作。

upstream 框架要求 `finalize_request` 必须实现，但通常它都是空函数，不需要特别的操作。^①

9.2.2 初始化

Nginx HTTP 框架接收到下游的连接就会调用 `ngx_http_create_request()` 创建 `ngx_http_request_t` 结构体，这时成员 `upstream` 默认是空指针，表示不使用 upstream 框架。

要启用 Nginx 的请求转发机制，就必须调用函数 `ngx_http_upstream_create()`，创建 upstream 对象。

函数 `ngx_http_upstream_create()` 的实现代码摘要如下：

```
//定义在 http/ngx_http_upstream.c
ngx_int_t ngx_http_upstream_create(ngx_http_request_t *r)
{
    ngx_http_upstream_t *u;

    u = r->upstream;                                     //upstream 对象

    u = ngx_palloc(r->pool, sizeof(ngx_http_upstream_t)); //创建对象

    r->upstream = u;                                     //指针赋值

    u->headers_in.content_length_n = -1;                 //初始化长度为无效值-1
    u->headers_in.last_modified_time = -1;

    return NGX_OK;
}
```

函数 `ngx_http_upstream_create()` 的工作很简单，只是创建了 upstream 对象，里面的很多字段都是无意义的 0，必须由之后的代码进一步设置。

① 目前官方所有 upstream 模块的 `finalize_request` 都是空函数，所以 upstream 框架实在有必要为它提供一个默认的空实现，或者检查空指针。很可惜，直到目前的 1.12.0 版仍然没有。

9.2.3 设置连接参数

upstream 机制的大部分连接参数由 `ngx_http_upstream_conf_t` 结构体决定，也就是 `ngx_http_upstream_t::conf` 成员。

设置基本参数

设置 upstream 参数的标准做法是在模块的配置数据结构里也定义一个 `ngx_http_upstream_conf_t` 成员，名字通常就是 `upstream`，在配置解析时使用指令设置里面的各个字段，最后在请求转发前直接把它赋值给 `ngx_http_upstream_t::conf`，即：

```
r->upstream->conf = &lcf->upstream; //设置 upstream 连接参数
```

在 `ngx_http_upstream_conf_t` 里必须要设置的是超时时间 `connect_timeout`、`send_timeout`、`read_timeout` 和缓冲区大小 `buffer_size` 这四个参数，因为从内存池里创建后这些参数的默认值都是 0，会让 upstream 框架无法正常工作。

设置缓冲区

`ngx_http_upstream_t` 结构里的 `buffering` 标志位指示 upstream 框架是否使用更多的缓冲区来接收上游的数据，但它与 `ngx_http_upstream_conf_t::buffering` 没有直接的关系，需要由我们自己来手工指定：

```
r->upstream->buffering = lcf->upstream.buffering; //设置是否使用更多的缓冲区
```

如果 `buffering` 值是 1，那么 upstream 框架会分配更多的缓冲区来接收数据，否则只使用固定的大小，即 `buffer_size`。

设置上游服务器地址

通常我们会在配置文件里使用 `upstream` 指令配置上游服务器的列表，使用函数 `ngx_http_upstream_add()` 就可以设置 `ngx_http_upstream_conf_t::upstream` 成员，upstream 框架会使用负载均衡算法在服务器列表里选择合适的服务器。

函数 `ngx_http_upstream_add()` 的声明是：

```
ngx_http_upstream_srv_conf_t *ngx_http_upstream_add(ngx_conf_t *cf,  
                                                    ngx_url_t *u, ngx_uint_t flags);
```

我们也可以直接使用 `ngx_http_upstream_t` 里的成员 `resolved`，它可以用原始 Socket API 的方式直接指定服务器的地址，例如：

```
sockaddr_in addr;
```

```
addr.sin_family = AF_INET;
addr.sin_port = htons(80);
addr.sin_addr.s_addr = inet_addr("127.0.0.1");

u->resolved = NgxPool(r).alloc<ngx_http_upstream_resolved_t>();

u->resolved->sockaddr = (sockaddr*)&addr;
u->resolved->socklen = sizeof(sockaddr_in);
u->resolved->naddrs = 1;
```

实际开发中这两种方式都可以采用,第一种方式的好处是可以在配置文件里定义多个服务器,并且使用负载均衡算法;而第二种方式的好处是用法灵活,可以根据请求的具体情况任意设置上游服务器。

9.2.4 启动连接

设置好 upstream 框架所需的回调函数和连接参数后,调用函数 `ngx_http_upstream_init()` 就可以启动 upstream 框架,开始与上游服务器异步交互。

`ngx_http_upstream_init()` 的声明是:

```
void ngx_http_upstream_init(ngx_http_request_t *r);
```

如果下游有请求体需要转发,那么可以把 `ngx_http_upstream_init()` 作为 `ngx_http_read_client_request_body()` 的回调函数,即:

```
ngx_http_read_client_request_body(r, ngx_http_upstream_init);
```

这样当 HTTP 框架接收完请求体数据后会回调 `ngx_http_upstream_init()`,启动 upstream 框架。

因为 upstream 框架发起了一个新的异步请求,所以通常请求的引用计数需要加 1,告诉 Nginx 框架不要销毁请求对象。`ngx_http_upstream_init()` 内部没有执行计数增加操作,这是特意的,因为 `ngx_http_read_client_request_body()` 函数里已经有了增加引用计数的代码,再增加会导致运行错误。

在启动 upstream 机制后,必须返回错误码 `NGX_DONE`,让 HTTP 处理引擎继续运行。

9.2.5 处理数据

upstream 框架调用 `create_request()` 得到请求数据后就会向上游发送请求,然后在 `ngx_http_upstream_process_header()` 里处理接收到的数据,代码摘要如下:

```

static void ngx_http_upstream_process_header(...)
{
    for ( ;; ) {
        rc = u->process_header(r); //接收数据解析响应头
        if (rc == NGX_AGAIN) {      //还未解析完响应头
            continue;               //继续循环处理
        }
        break;                      //返回 NGX_OK, 响应头解析完毕
    }

    if (!r->subrequest_in_memory) { //检查标志位
        ngx_http_upstream_send_response(r, u); //直接转发响应体数据
        return;
    }

    if (u->input_filter == NULL) {    //是否有 input_filter 回调函数
        u->input_filter_init = ...;    //没有则使用默认函数处理
        u->input_filter = ...;
        u->input_filter_ctx = r;
    }

    if (u->input_filter_init(         //过滤处理初始化
        u->input_filter_ctx) == NGX_ERROR) {
        ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
        return;
    }

    if (u->input_filter(              //过滤处理响应体数据
        u->input_filter_ctx, n) == NGX_ERROR) {
        ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
        return;
    }
}

```

它首先调用 `process_header()` 解析响应头, 当 `process_header()` 返回 `NGX_OK` 时就表示响应头接收完毕, 之后的数据都是响应体。如果 `subrequest_in_memory` 标志位是 0, 那么就调用 `ngx_http_upstream_send_response()` 直接把数据转发给下游, 否则调用模块提供的 `input_filter` 处理响应体。

9.3 负载均衡机制

当 Nginx 作为反向代理时, 它连接的上游服务器可能不是简单的一台服务器, 而是一个服务器集群, 这就需要使用某种算法在集群里选择一台适当的服务器, 尽量把请求均衡地转发

给集群里的每台服务器，充分利用系统资源。

目前已经有相当成熟的负载均衡算法，例如轮询 (round robin)、IP 散列 (ip hash)、一致性散列 (consistent hash) 等，Nginx 在 upstream 框架里抽象出了 load-balance 模块，封装了这些算法。

load-balance 模块是一种更为特殊的 http 模块，它不处理任何数据，只是实现选择服务器的算法，所以它的运行机制与 handler、filter、upstream 模块有很大不同。

load-balance 模块的配置指令只能出现在 upstream 块里，在定义指令属性时要使用宏 NGX_HTTP_UPS_CONF。

9.3.1 结构定义

load-balance 模块涉及较多的数据结构，本节只介绍几个最关键的。

初始化结构体

结构体 ngx_http_upstream_peer_t 定义了 load-balance 模块的入口，有两个回调函数，用来初始化 load-balance 模块：

```
//定义在 http/nginx_http_upstream.h
typedef ngx_int_t (*ngx_http_upstream_init_pt) (ngx_conf_t *cf,
                                                ngx_http_upstream_srv_conf_t *us);
typedef ngx_int_t (*ngx_http_upstream_init_peer_pt) (ngx_http_request_t *r,
                                                    ngx_http_upstream_srv_conf_t *us);

typedef struct {
    ngx_http_upstream_init_pt    init_upstream; //配置解析时初始化
    ngx_http_upstream_init_peer_pt init;        //请求时初始化
    void*                        data;           //服务器列表指针
} ngx_http_upstream_peer_t;
```

它位于 upstream 框架的配置数据结构 ngx_http_upstream_srv_conf_t 里：

```
struct ngx_http_upstream_srv_conf_s {
    ngx_http_upstream_peer_t    peer; //初始化结构体
    ...                          //其他成员
};
```

每一个 upstream 配置块对应一个 ngx_http_upstream_srv_conf_t 结构，也就是说，每一个 upstream 块只能使用一种负载均衡算法。

连接结构体

结构体 `ngx_peer_connection_t` 定义了 load-balance 模块实现负载均衡算法的回调函数和相关字段，定义摘要如下：

```
//定义在 event/nginx_event_connect.h
typedef struct ngx_peer_connection_s  ngx_peer_connection_t;

typedef ngx_int_t (*ngx_event_get_peer_pt)(ngx_peer_connection_t *pc,
                                           void *data);
typedef void (*ngx_event_free_peer_pt)(ngx_peer_connection_t *pc,
                                       void *data, ngx_uint_t state);

struct ngx_peer_connection_s {
    ngx_connection_t*      connection;           //TCP 连接对象

    struct sockaddr*       sockaddr;             //socket 地址
    socklen_t              socklen;
    ngx_str_t*             name;

    ngx_uint_t             tries;                //重试次数

    ngx_event_get_peer_pt  get;                  //执行算法，获取服务器地址
    ngx_event_free_peer_pt free;                //获取服务器地址后的更新操作
    void*                  data;                //get 需要使用的数据

    ngx_log_t*             log;                  //日志对象
    unsigned               cached:1;            //是否使用磁盘缓存
};
```

它是 upstream 框架的核心数据结构 `ngx_http_upstream_t` 的成员：

```
struct ngx_http_upstream_s {
    ngx_peer_connection_t  peer;                //使用算法连接服务器
    ...                                           //其他成员
};
```

注意它的名字也是 `peer`，小心不要和 `ngx_http_upstream_srv_conf_t::peer` 弄混了，后者在配置解析阶段起作用，只用于初始化算法。

服务器信息

结构体 `ngx_http_upstream_server_t` 存放了 upstream 配置块里每个服务器的信息，供模块计算时参考，它与 upstream 块的 `sever` 指令完全对应：

```
//定义在 http/nginx_http_upstream.h
```

```
typedef struct {
    ngx_str_t          name;           //服务器名字
    ngx_addr_t*        addrs;          //服务器对应的地址数组
    ngx_uint_t         naddrs;         //地址数组的长度
    ngx_uint_t         weight;         //权重
    ngx_uint_t         max_conns;      //允许的最大连接数
    ngx_uint_t         max_fails;      //允许的最大失败次数
    time_t             fail_timeout;   //失败的时间区间
    ngx_msec_t         slow_start;     //服务器的恢复时间

    unsigned            down:1;         //服务器是否下线
    unsigned            backup:1;      //是否是备用服务器
} ngx_http_upstream_server_t;
```

在配置解析时, Nginx 会使用函数 `ngx_parse_url()` 得到服务器对应的地址, 填充 `ngx_http_upstream_server_t` 结构。因为一个域名可能会对应多个 IP 地址, 所以成员 `addrs` 是一个数组。

`ngx_http_upstream_srv_conf_t` 使用动态数组保存了一个 `upstream` 配置块里所有的服务器信息:

```
struct ngx_http_upstream_srv_conf_s {
    ngx_array_t*      servers;         //ngx_http_upstream_server_t 数组
    ...               //其他成员
};
```

IP 地址列表

`upstream` 框架实现了 `round-robin` 算法作为默认的负载均衡算法, 使用了三个名字非常相像的结构体来管理上游服务器 IP, 是其他算法的基础。

`ngx_http_upstream_rr_peer_t` 与每个服务器的具体 IP 地址一一对应, 摘要如下:

```
//定义在 http/nginx_http_upstream_round_robin.h
typedef struct {
    struct sockaddr* sockaddr;         //可连接的一个 IP 地址
    socklen_t        socklen;         //sockaddr 结构体的长度
    ngx_str_t        name;            //地址的名字
    ngx_str_t        server;          //服务器的名字
    ...               //其他成员
} ngx_http_upstream_rr_peer_t;
```

结构体 `ngx_http_upstream_rr_peers_t` (名字多了一个 “s”) 管理 IP 地址列表:

```
typedef struct ngx_http_upstream_rr_peers_s ngx_http_upstream_rr_peers_t;
```

```
struct ngx_http_upstream_rr_peers_s {  
    ngx_uint_t          number;           //服务器数量,即 peer 的长度  
    unsigned            single:1;         //只有一台服务器时优化处理  
    ngx_str_t*          name;             //upstream 块的名字  
  
    ngx_http_upstream_rr_peers_t* next;   //备用 (backup) 服务器 IP 列表  
    ngx_http_upstream_rr_peer_t* peer;    //在用 (非 backup) 服务器 IP 列表  
};
```

peer 里面保存的是所有在用服务器的 IP 地址信息,而备用服务器保存在 next.peer 成员里。

结构体 ngx_http_upstream_rr_peer_data_t 是负载均衡算法使用的数据结构,从 peers.peer 就可以获得可用的 IP 地址列表:

```
typedef struct {  
    ngx_http_upstream_rr_peers_t* peers;           //IP 地址列表  
    ngx_uint_t          current;                   //round robin 算法参数  
    uintptr_t*          tried;                    //重试 bit 数组  
    uintptr_t  
} ngx_http_upstream_rr_peer_data_t;
```

结构体关系图

load-balance 模块使用的数据结构较多,命名也不够清晰,显得有些杂乱,图 9-2 中的 UML 图粗略地描述了它们之间的关系,可以帮助我们理解。

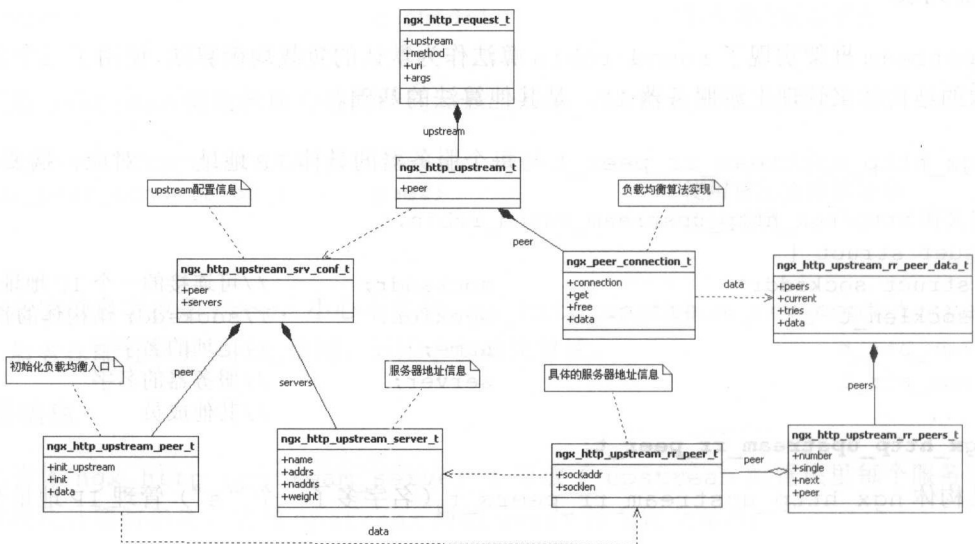


图 9-2 load-balance 模块相关类图

9.3.2 初始化模块入口

load-balance 模块必须要定义一个配置指令以启用算法，在指令解析函数里设置入口函数指针 `peer.init_upstream`，并决定服务器允许哪些附加配置参数，基本的代码是：

```
uscf = ngx_http_conf_get_module_srv_conf(cf, ngx_http_upstream_module);
uscf->peer.init_upstream = ...;           //设置入口函数
uscf->flags = ...;                        //设置服务器配置参数
```

`uscf->flags` 决定了 `upstream` 配置块的 `server` 指令可以使用哪些参数，取值是如下宏的逻辑或：

```
#define NGX_HTTP_UPSTREAM_CREATE      0x0001    //不允许重复服务器
#define NGX_HTTP_UPSTREAM_WEIGHT     0x0002    //允许 weight 参数
#define NGX_HTTP_UPSTREAM_MAX_FAILS  0x0004    //允许 max_falis 参数
#define NGX_HTTP_UPSTREAM_FAIL_TIMEOUT 0x0008   //允许 fail_timeout 参数
#define NGX_HTTP_UPSTREAM_DOWN       0x0010    //允许 down 参数
#define NGX_HTTP_UPSTREAM_BACKUP     0x0020    //允许 backup 参数
```

如果 `flags` 值是 0，那么 `server` 指令就不能使用任何附加配置参数。

HTTP 框架在配置文件解析完成后会调用 `ngx_http_upstream_module` 模块的 `ngx_http_upstream_init_main_conf()` 函数进行初始化，在这里 `upstream` 框架会调用 `peer.init_upstream` 初始化每个上游服务器列表所使用的 load-balance 模块，代码摘要如下：

```
static char *
ngx_http_upstream_init_main_conf(ngx_conf_t *cf, void *conf)
{
    ngx_http_upstream_main_conf_t *umcf = conf;           //main 配置
    ngx_http_upstream_srv_conf_t **uscfp;                //服务器配置数组

    uscfp = umcf->upstreams.elts;                          //得到所有 upstream 块

    for (i = 0; i < umcf->upstreams.nelts; i++) {          //遍历数组

        init = uscfp[i]->peer.init_upstream ?              //是否有模块初始化函数
            uscfp[i]->peer.init_upstream:                  //使用算法的初始化函数
            ngx_http_upstream_init_round_robin;             //默认使用 round robin

        if (init(cf, uscfp[i]) != NGX_OK) {                //执行初始化函数
            return NGX_CONF_ERROR;
        }
    }
}
```

```
...
}
```

9.3.3 初始化地址列表

虽然我们可以实现任意的负载均衡算法，但 round robin 算法是 Nginx 负载均衡的基础，load-balance 模块必须使用函数 `ngx_http_upstream_init_round_robin()` 初始化服务器 IP 地址列表，然后再基于这个列表实现特定的算法。

在 `peer.init_upstream` 函数里首先需要调用 `ngx_http_upstream_init_round_robin()` 函数，从配置文件里得到所有的服务器 IP 地址，构建出可用的 IP 地址列表。它的核心代码摘要如下：^①

```
ngx_int_t
ngx_http_upstream_init_round_robin(ngx_conf_t *cf,
                                   ngx_http_upstream_srv_conf_t *us)
{
    server = us->servers->elts;                //服务器数组首地址

    us->peer.init =                             //默认使用 round robin
        ngx_http_upstream_init_round_robin_peer;

    n = 0;                                     //计算地址的总数
    w = 0;                                     //计算总权重
    for (i = 0; i < us->servers->nelts; i++) {   //遍历服务器数组
        if (server[i].backup) {                 //计算在用服务器
            continue;
        }

        n += server[i].naddrs;                  //计算地址的总数
        w += server[i].naddrs * server[i].weight; //计算总权重
    }

    peers = ngx_palloc(cf->pool, ...);          //创建 IP 列表数据结构
    peer = ngx_palloc(cf->pool, sizeof(ngx_http_upstream_rr_peer_t) * n);

    peers->single = (n == 1);                   //是否只有一个服务器
    peers->number = n;                           //服务器数量
    peers->weighted = (w != n);                 //是否加权
    peers->total_weight = w;                    //总权重
    peers->name = &us->host;                    //upstream 块的名字
}
```

^① 这里省略了不使用 upstream 块直接给出上游服务器地址的处理代码。

```

n = 0;

for (i = 0; i < us->servers->nelts; i++) {           //遍历服务器数组
    if (server[i].backup) {                          //只处理在用服务器
        continue;
    }

    for (j = 0; j < server[i].naddrs; j++) {         //遍历服务器的每个 IP 地址
        peer[n].sockaddr = server[i].addrs[j].sockaddr; //socket 地址
        peer[n].socklen = server[i].addrs[j].socklen;  //socket 结构长度
        peer[n].name = server[i].addrs[j].name;        //地址的名字
        ...                                           //权重等其他参数
        n++;
    }
}

us->peer.data = peers;                             //添加到配置数据结构里

backup = ...;                                       //备用服务器列表
peers->next = backup;                               //挂到 next 指针下

...                                                //处理过程与在用服务器类似

return NGX_OK;
}

```

函数首先设置了 `peer.init` 函数指针，默认使用 `round robin` 算法，然后创建 IP 列表数据结构 `peers` 和 IP 地址信息数组 `peer`。

填充完 IP 地址列表信息之后是一个关键操作：

```
us->peer.data = peers; //关键操作!
```

这样，构造出的 IP 地址列表就又放回到了配置结构数据的 `us->peer.data` 里，在后续的请求处理时就可以直接访问。

调用 `ngx_http_upstream_init_round_robin()` 后，`upstream` 框架使用的算法是 `round robin`，所以 `load-balance` 模块需要改写 `peer.init` 函数指针为自己的初始化函数，即：

```

if (ngx_http_upstream_init_round_robin(cf, us) != NGX_OK) {
    return NGX_ERROR;
}

us->peer.init = ...; //设置模块的算法初始化函数

```

9.3.4 初始化算法

在启动函数 `ngx_http_upstream_init()` 里 `upstream` 框架会查找匹配的 `upstream` 块配置, 并调用 `peer.init` 函数初始化负载均衡算法。

`peer.init` 函数的主要工作是构造模块自己的 `ngx_http_upstream_rr_peer_data_t` 对象, 从 `us->peer.data` 得到 IP 地址列表, 用于后续的算法计算, 这也可以利用 `ngx_http_upstream_init_round_robin_peer()` 函数来完成。

`ngx_http_upstream_init_round_robin_peer()` 函数代码摘要如下:

```
ngx_int_t
ngx_http_upstream_init_round_robin_peer(ngx_http_request_t *r,
                                         ngx_http_upstream_srv_conf_t *us)
{
    ngx_http_upstream_rr_peer_data_t *rrp;

    rrp = r->upstream->peer.data;           //使用 upstream 里的 peer.data

    if (rrp == NULL) {                       //空指针则分配内存
        rrp = ngx_palloc(r->pool, sizeof(ngx_http_upstream_rr_peer_data_t));
        r->upstream->peer.data = rrp;
    }

    rrp->peers = us->peer.data;              //得到配置里的 IP 地址列表
    rrp->current = NULL;

    r->upstream->peer.get = ngx_http_upstream_get_round_robin_peer;
    r->upstream->peer.free = ngx_http_upstream_free_round_robin_peer;

    return NGX_OK;
}
```

这段代码里利用了结构体 `ngx_peer_connection_t` 里的 `data` 成员, 使用它存放了 IP 地址列表数据。

之后 `load-balance` 模块必须要设置 `get/free` 回调函数, 供 `upstream` 框架在连接上游服务器时调用, 否则还会是默认的 `round robin` 算法。

9.3.5 执行算法

`upstream` 框架使用函数 `ngx_http_upstream_connect()` 连接上游服务器, 这时会调用 `peer.get` 函数指针, 使用负载均衡算法从 IP 地址列表里选择一个上游服务器:

```
//定义在 http/nginx_http_upstream.c
static void ngx_http_upstream_connect(...)
{
    ...
    rc = ngx_event_connect_peer(&u->peer);           //连接上游服务器
    ...
}

//定义在 event/nginx_event_connect.c
ngx_int_t ngx_event_connect_peer(ngx_peer_connection_t *pc)
{
    ...
    rc = pc->get(pc, pc->data);                       //调用 get 函数指针
    if (rc != NGX_OK) {
        return rc;
    }

    s = ngx_socket(...);                             //创建 socket 对象
    ...
}
```

如果请求结束或者上游服务器无法连接，upstream 框架会执行 `peer.free` 函数指针，使用 `state` 参数传递错误原因，模块可以在这里调整 IP 地址列表和算法的内部状态，为下一次运行做好准备。

`peer.free` 函数不是必须的，但不能是空指针，如果算法不需要 `free` 操作，upstream 会使用 `round robin` 默认的 `free` 函数。

9.4 源码分析

本节简要研究 `ngx_http_memcached_module` 和 `ngx_http_upstream_ip_hash_module` 这两个模块的源码，它们分别实现了连接上游 Memcached 服务器和 IP 散列负载均衡算法。

9.4.1 ngx_http_memcached_module

`ngx_http_memcached_module` 是一个 upstream 模块，它可以连接 Memcached 服务器，读取缓存数据（不提供写入操作）。

配置数据结构

`ngx_http_memcached_module` 在配置数据结构里定义了 upstream 的连接参数：

```
typedef struct {
    ngx_http_upstream_conf_t    upstream;           //upstream 连接参数
    ngx_int_t                    index;
    ngx_uint_t                   gzip_flag;
} ngx_http_memcached_loc_conf_t;
```

配置指令

ngx_http_memcached_module 的核心指令是启动 upstream 框架的 memcached_pass，其他指令都是设置各种连接参数：

```
static ngx_command_t  ngx_http_memcached_commands[] = {

    { ngx_string("memcached_pass"),           //启动 upstream 框架
      NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF|NGX_CONF_TAKE1,
      ngx_http_memcached_pass,                //指令解析函数
      NGX_HTTP_LOC_CONF_OFFSET,
      0,
      NULL },

    { ngx_string("memcached_connect_timeout"), //连接超时时间
      NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|
        NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
      ngx_conf_set_msec_slot,
      NGX_HTTP_LOC_CONF_OFFSET,
      offsetof(ngx_http_memcached_loc_conf_t, upstream.connect_timeout),
      NULL },

    ...                                       //其他指令
    ngx_null_command
};
```

设置处理函数

函数 ngx_http_memcached_pass() 解析配置指令，设置模块的 content handler：

```
static char *
ngx_http_memcached_pass(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    ngx_http_memcached_loc_conf_t *mlcf = conf;
    value = cf->args->elts;           //得到指令字符串

    ngx_memzero(&u, sizeof(ngx_url_t));

    u.url = value[1];                  //解析 url
    u.no_resolve = 1;
```

```

mlcf->upstream.upstream = ngx_http_upstream_add(cf, &u, 0);
if (mlcf->upstream.upstream == NULL) {
    return NGX_CONF_ERROR;
}

clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
clcf->handler = ngx_http_memcached_handler;    //注册处理函数
}

```

启动 upstream 机制

函数 `ngx_http_memcached_handler()` 设置 upstream 回调函数，启动 upstream 机制，代码摘要如下：

```

static ngx_int_t
ngx_http_memcached_handler(ngx_http_request_t *r)
{
    if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD))) {    //检查请求方法
        return NGX_HTTP_NOT_ALLOWED;
    }

    rc = ngx_http_discard_request_body(r);                //丢弃请求体

    if (ngx_http_upstream_create(r) != NGX_OK) {          //初始化 upstream
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }

    mlcf = ngx_http_get_module_loc_conf(r, ngx_http_memcached_module);
    u->conf = &mlcf->upstream;                            //设置连接参数

    //设置回调函数
    u->create_request = ngx_http_memcached_create_request;
    u->reinit_request = ngx_http_memcached_reinit_request;
    u->process_header = ngx_http_memcached_process_header;
    u->abort_request = ngx_http_memcached_abort_request;
    u->finalize_request = ngx_http_memcached_finalize_request;

    u->input_filter_init = ngx_http_memcached_filter_init;
    u->input_filter = ngx_http_memcached_filter;
    u->input_filter_ctx = ctx;

    r->main->count++;                                     //增加引用计数
    ngx_http_upstream_init(r);                          //启动 upstream
}

```

```
return NGX_DONE; //必须返回 NGX_DONE
}
```

因为访问 Memcached 服务器不需要请求体，所以函数里丢弃了请求体，增加引用计数后直接启动了 upstream 机制。

回调函数

ngx_http_memcached_module 解析数据的回调函数代码较为复杂，与 Memcached 协议关系密切，故这里不列出具体的代码，只介绍实现的基本功能。

函数 ngx_http_memcached_create_request() 从变量 \$memcached_key 获取 key，然后构造 get 命令。

函数 ngx_http_memcached_process_header() 解析 Memcached 数据，当找到换行符时表示响应头解析完毕，设置 u->headers_in.content_length_n、u->headers_in.status_n 和 u->state->status 等响应信息。

函数 ngx_http_memcached_filter() 处理响应数据，主要的工作是去掉 Memcached 数据末尾的“END”标识符，并把数据存放在 u->out_bufs 链表里。

函数 ngx_http_memcached_finalize_request() 是一个空函数，里面只记录了一条 debug 日志。

9.4.2 ngx_http_upstream_ip_hash_module

ngx_http_upstream_ip_hash_module 是一个 load-balance 模块，它实现了 IP 散列负载均衡算法。

配置指令

ngx_http_upstream_ip_hash_module 没有配置数据结构，只用一个 ip_hash 指令来启用算法，指令数组是：

```
static ngx_command_t ngx_http_upstream_ip_hash_commands[] = {
    { ngx_string("ip_hash"),
      NGX_HTTP_UPS_CONF|NGX_CONF_NOARGS,
      ngx_http_upstream_ip_hash,
      0,
      0,
      NULL },
    ngx_null_command
}
```



```
};
```

设置模块入口

函数 `ngx_http_upstream_ip_hash()` 解析配置指令，设置算法的入口函数 `peer.init_upstream`:

```
static char *
ngx_http_upstream_ip_hash(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    uscf = ngx_http_conf_get_module_srv_conf(cf, ngx_http_upstream_module);

    if (uscf->peer.init_upstream) { //已设置则配置出错
        ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
                           "load balancing method redefined");
    }

    uscf->peer.init_upstream = ngx_http_upstream_init_ip_hash;
    uscf->flags = ...; //设置允许的配置参数

    return NGX_CONF_OK;
}
```

初始化 IP 地址列表

函数 `ngx_http_upstream_init_ip_hash()` 调用 `round robin` 函数初始化 IP 地址列表，然后设置自己的算法初始化函数 `peer.init`:

```
static ngx_int_t
ngx_http_upstream_init_ip_hash(ngx_conf_t *cf,
                               ngx_http_upstream_srv_conf_t *us)
{
    if (ngx_http_upstream_init_round_robin(cf, us) != NGX_OK) {
        return NGX_ERROR;
    }

    us->peer.init = ngx_http_upstream_init_ip_hash_peer; //算法初始化函数

    return NGX_OK;
}
```

初始化算法

`ngx_http_upstream_ip_hash_module` 定义了专用的算法数据对象 `ngx_http_upstream_ip_hash_peer_data_t`，必须要注意的是它利用了 C 语言的平坦内存模型，把

ngx_http_upstream_rr_peer_data_t 作为它的第一个成员，这样 round robin 算法可以忽略后面的成员，把它当作是 ngx_http_upstream_rr_peer_data_t 来操作：

```
typedef struct {
    ngx_http_upstream_rr_peer_data_t  rrp;           //算法数据对象，必须是第一个
    ...                                     //其他成员
    ngx_event_get_peer_pt  get_rr_peer;             //round robin 算法
} ngx_http_upstream_ip_hash_peer_data_t;
```

函数 ngx_http_upstream_init_ip_hash_peer() 创建算法数据对象，并使用 round robin 算法得到配置里的 IP 地址列表，然后再设置 get 回调：

```
static ngx_int_t
ngx_http_upstream_init_ip_hash_peer(ngx_http_request_t *r,
    ngx_http_upstream_srv_conf_t *us)
{
    ngx_http_upstream_ip_hash_peer_data_t  *iphp;

    iphp = ngx_palloc(...);                       //创建算法数据结构
    r->upstream->peer.data = &iphp->rrp;           //设置 peer.data

    //得到配置里的 IP 地址列表
    if (ngx_http_upstream_init_round_robin_peer(r, us) != NGX_OK) {
        return NGX_ERROR;
    }

    r->upstream->peer.get = ngx_http_upstream_get_ip_hash_peer;
    iphp->get_rr_peer = ngx_http_upstream_get_round_robin_peer;

    return NGX_OK;
}
```

IP 散列算法没有内部状态，所以它不需要设置 free 回调函数。

执行算法

函数 ngx_http_upstream_get_ip_hash_peer 是 IP 散列算法的具体实现，根据 IP 地址计算散列值，然后从列表中选择一个地址：

```
static ngx_int_t
ngx_http_upstream_get_ip_hash_peer(ngx_peer_connection_t *pc, void *data)
{
    ngx_http_upstream_ip_hash_peer_data_t  *iphp = data;

    if (iphp->tries > 20 || iphp->rrp.peers->single) { //重试次数过多
```

```

    return iphp->get_rr_peer(pc, &iphp->rrp);           //退化为 round robin
}

pc->cached = 0;                                         //不使用磁盘缓存
pc->connection = NULL;                                 //尚未连接

...                                                    //算法计算散列值
peer = &iphp->rrp.peers->peer[p];                     //得到一个地址

pc->sockaddr = peer->sockaddr;                          //设置连接地址
pc->socklen = peer->socklen;
pc->name = &peer->name;

return NGX_OK;                                         //成功获得一个上游地址
}

```

9.5 C++封装

本节实现四个C++类,封装Nginx的upstream机制,方便开发upstream模块和load-balance模块。

9.5.1 NgxUpstream

ngx_http_upstream_t结构体是upstream框架的核心数据结构,类NgxUpstream是它的C++代理,定义如下:

```

class NgxUpstream final : public NgxWrapper<ngx_http_upstream_t>
{
public:
    typedef NgxWrapper<ngx_http_upstream_t> super_type;
    typedef NgxUpstream this_type;

public:
    NgxUpstream(ngx_http_upstream_t* u)                //直接代理结构体
        :super_type(u)
    {}

    NgxUpstream(ngx_http_request_t* r):                 //获取 request_t 的成员
        super_type(r->upstream)
    {}

    ~NgxUpstream() = default;

    ...                                                //其他成员函数, 见后
};

```

基本操作

类 `NgxUpstream` 使用成员函数 `headers()` 和 `state()` 操作常用的两个成员变量 `headers_in` 和 `state`。注意在原数据结构里这两个成员的类型不同（分别是对象实体和指针），但函数统一返回了引用的形式，有利于客户编码：

```
public:
    ngx_http_upstream_headers_in_t& headers() const //获取上游返回的头部信息
    {
        return get()->headers_in; //返回引用
    }

    ngx_http_upstream_state_t& state() const //获取上游的状态信息
    {
        return *get()->state; //返回引用
    }
```

设置配置参数

成员函数 `conf()` 和 `buffering()` 用来设置 upstream 框架连接上游的基本参数：

```
public:
    void conf(ngx_http_upstream_conf_t& cf) const //设置上游的连接参数
    {
        get()->conf = &cf; //设置配置结构体
        get()->buffering = cf.buffering; //缓冲标志位
    }

    void buffering(bool flag) const //单独设置是否使用缓冲
    {
        get()->buffering = flag; //缓冲标志位
    }
```

设置请求数据

构造发送给上游服务器的请求是启动 upstream 机制时非常关键的功能，成员函数 `request()` 简化了这一操作，把数据挂在链表的末尾，允许多次调用，分步构造出最终的字符串：

```
public:
    void request(ngx_chain_t* bufs) const //构造请求数据
    {
        if(!get()->request_bufs) //无请求数据则直接赋值
        {
            get()->request_bufs = bufs;
        }
```

```

    return;
}

NgxChain ch = get()->request_bufs;           //得到数据链表
ch.append(bufs);                             //添加到链表末尾
}

```

9.5.2 NgxUpstreamHelper

NgxUpstream 代理了 ngx_http_upstream_t 结构体，但它自身无法完成初始化、设置回调函数、启动等操作，需要用辅助类 NgxUpstreamHelper 来实现。

类定义

NgxUpstreamHelper 使用模板参数，在编译期确定了 upstream 机制必须实现的四个回调函数，定义如下：

```

template<
    ngx_int_t(*create_request)(ngx_http_request_t*) = nullptr,
    ngx_int_t(*reinit_request)(ngx_http_request_t *r) = nullptr,
    ngx_int_t(*process_header)(ngx_http_request_t*) = nullptr,
    void(*finalize_request)(ngx_http_request_t*,ngx_int_t) = nullptr
>
class NgxUpstreamHelper final : public NgxWrapper<ngx_http_request_t>
{
public:
    typedef NgxWrapper<ngx_http_request_t>    super_type;    //简化类型定义
    typedef NgxUpstreamHelper                  this_type;
    typedef NgxUpstream                        upstream_type;

public:
    NgxUpstreamHelper(ngx_http_request_t* r):
        super_type(r), m_upstream(create())                //构造时创建 upstream 对象
    {}

    ~NgxUpstreamHelper() = default;

public:
    const upstream_type& upstream() const                  //获取 upstream 对象
    {
        return m_upstream;
    }

private:
    upstream_type m_upstream;                               //upstream 对象

private:
    static void default_finalize_request(ngx_http_request_t* r,ngx_int_t rc)
    {

```

```

    NgxLogDebug(r).print("default_finalize_request");
}
...
};
```

//其他成员函数，见后

成员 `m_upstream` 代理了 `ngx_http_request_t` 结构体里的 `upstream`，它会在构造函数里调用 `create()` 函数初始化。

通常 `upstream` 模块的 `finalize_request()` 都是空操作，所以 `NgxUpstream-Helper` 提供了一个默认的实现——如果客户代码不提供 `finalize_request` 指针，就使用内置的静态成员函数 `default_finalize_request()`。

初始化 upstream

成员函数 `create()` 调用函数 `ngx_http_upstream_create()`，初始化 `ngx_http_request_t` 里的 `upstream` 成员：

```

private:
    ngx_http_upstream_t* create() const                //初始化 upstream 对象
    {
        if(!get()->upstream)                          //检查 upstream 指针
        {
            auto rc = ngx_http_upstream_create(get()); //创建 upstream 对象

            NgxException::require(                      //错误码检查
                rc == NGX_OK, NGX_HTTP_INTERNAL_SERVER_ERROR);
        }

        return get()->upstream;                        //返回 upstream 对象
    }
```

配置 upstream

成员函数 `conf()` 可以直接设置 `upstream` 框架的连接参数，内部调用了 `NgxUpstream` 的 `conf()` 函数：

```

void conf(ngx_http_upstream_conf_t& cf) const
{
    upstream().conf(cf);                               //调用 NgxUpstream 的 conf()
}
```

启动 upstream

启动函数 `start()` 使用一个参数来决定是否读取客户端的请求体数据，它首先设置四个基本的回调函数，然后根据参数使用不同的方式调用 `ngx_http_upstream_init()` 启动

upstream 框架，最后返回 NGX_DONE:

```
public:
    ngx_int_t start(bool read_body = false) const //启动 upstream 框架
    {
        if(!upstream()->create_request) //设置回调函数
        {
            upstream()->create_request = create_request; //创建请求函数指针
            upstream()->reinit_request = reinit_request; //重新初始化函数指针
            upstream()->process_header = process_header; //处理响应头函数指针
            upstream()->finalize_request = //检查函数指针是否为空
                finalize_request?finalize_request: //不为空则使用函数指针
                &this_type::default_finalize_request; //否则使用默认的空函数
        }

        if(read_body) //是否要读取请求体
        {
            auto rc = //读取完请求体后再启动
                ngx_http_read_client_request_body(
                    get(), ngx_http_upstream_init); //upstream_init 作为回调

            NgxException::fail(rc >= NGX_HTTP_SPECIAL_RESPONSE);
        }
        else //不读取请求体，直接启动
        {
            auto rc = ngx_http_discard_request_body(get()); //丢弃请求体
            NgxException::require(rc);

            ++get()->main->count; //增加主请求的引用计数
            ngx_http_upstream_init(get()); //启动 upstream 框架
        }

        return NGX_DONE; //返回 NGX_DONE，继续引擎
    }
}
```

这段代码中需要注意的是 `read_body==false` 时的处理，我们必须要增加主请求的引用计数，告诉 Nginx 框架不要销毁请求对象。

9.5.3 NgxHttpUpstreamModule

在设置 load-balance 模块入口函数时我们需要访问 `ngx_http_upstream_module` 的配置，所以可以为它实现一个专门的类。

类定义

NgxHttpUpstreamModule 代理了 ngx_http_upstream_module，只要把模块的配置结构类型作为模板参数传递给基类 NgxModule 就可以复用便捷访问配置数据的能力，定义如下：

```
class NgxHttpUpstreamModule final :
    public NgxModule<void,                                //注意没有 loc 配置
        ngx_http_upstream_srv_conf_t,
        ngx_http_upstream_main_conf_t>
{
public:
    typedef NgxModule<...>          super_type;        //简化类型定义
    typedef NgxHttpUpstreamModule  this_type;
public:
    NgxHttpUpstreamModule() : super_type(ngx_http_upstream_module)
    {}
    ~NgxHttpUpstreamModule() = default;
public:
    static NgxHttpUpstreamModule& instance()            //单件访问点
    {
        static NgxHttpUpstreamModule m;
        return m;
    }
    ...                                                //其他成员函数，见后
};
```

初始化算法入口

对于 load-balance 模块来说，我们需要修改配置里的 init_upstream 函数指针，让 upstream 框架使用 load-balance 模块定义的算法：

```
public:
    template<typename F>
    static void init(ngx_conf_t* cf, F f, ngx_uint_t flags = 0) const
    {
        auto& uscf = instance().conf().srv(cf);          //获取配置数据

        NgxException::fail(uscf.peer.init_upstream)      //已经设置则错误

        uscf.peer.init_upstream = f;                    //设置回调函数
        uscf.flags = flags;                              //设置算法参数
    }
```


9.5.4 NgxLoadBalance

NgxLoadBalance 类封装了 load-balance 模块的初始化操作。

类定义

NgxLoadBalance 类定义如下：

```
template<typename T,                                //算法数据结构
        ngx_event_get_peer_pt  get_peer,           //执行算法函数指针
        ngx_event_free_peer_pt  free_peer = nullptr,
        ngx_http_upstream_rr_peer_data_t T::* ptr = &T::rrp>
class NgxLoadBalance final
{
    typedef ngx_http_upstream_rr_peer_data_t peer_data_t;
    typedef ngx_http_upstream_srv_conf_t     srv_conf_t;
public:
    ...                                           //其他成员函数，见后
};
```

NgxLoadBalance 类有四个模板参数。T 是算法需要使用的 data，它里面的 rrp 成员是 round robin 算法使用的数据结构，可以用成员变量指针来访问。

初始化 IP 地址列表

init() 函数调用 ngx_http_upstream_init_round_robin() 初始化 IP 地址列表，并设置算法初始化函数：

```
public:
    static void init(ngx_conf_t *cf, srv_conf_t* us,
                    ngx_http_upstream_init_peer_pt init_peer)
    {
        auto rc = ngx_http_upstream_init_round_robin(cf, us);
        NgxException::require(rc);

        us->peer.init = init_peer;                //设置算法初始化函数
    }
```

初始化算法

在初始化算法时，我们需要创建算法数据结构，并设置 get/free 函数：

```
public:
    static T& init(ngx_http_request_t* r, srv_conf_t* us)
    {
        auto& peer_data = *NgxPool(r).alloc<T>(); //创建算法数据结构
    }
```

```

r->upstream->peer.data = &(peer_data.*ptr);    //设置算法数据指针

auto rc = ngx_http_upstream_init_round_robin_peer(r, us);
NgxException::require(rc);

r->upstream->peer.get = get_peer;                //设置 get 函数
r->upstream->peer.free = free_peer?free_peer:    //设置 free 函数
    r->upstream->peer.free;

return peer_data;                               //返回算法数据结构
}

```

9.6 开发 upstream 模块

完善的 upstream 模块要处理很多琐碎细节，本节的 upstream 模块仅作为示范，只有最基本的转发功能，读者可以以此为基础实现功能更完善的模块。

9.6.1 模块设计

假设有这样一个后端的 TCP echo 服务，它接受 4 个字节的请求，返回的数据是：2 字节长度信息+原数据+时间戳（这个服务的实现代码在 19.7.5 节），我们为它实现一个 upstream 模块，设计如下：

- 模块名是 ndg_upstream_module;
- 使用 content handler 方式注册处理函数;
- 使用指令 ndg_upstream_pass 启动转发处理;
- upstream 框架上游的连接参数均内部确定，不在配置文件里设置;
- 仅支持 GET 方法，转发 \$args，如长度不足 4 个字符，使用默认的“xxxx”;
- 解析响应头，把收到的数据转发到下游。

9.6.2 配置信息类

与 ngx_http_memcached_module 等模块的实现类似，我们在模块的配置数据结构里添加 ngx_http_upstream_conf_t 成员，用来设置 upstream 框架的连接参数：

```

class NdgUpstreamConf final                                //配置信息类
{
public:
    typedef NdgUpstreamConf this_type;

```

```

public:
    ngx_http_upstream_conf_t upstream;           //upstream 配置参数
public:
    static void* create(ngx_conf_t* cf)         //创建配置结构体
    {
        auto& c = *NgxPool(cf).alloc<this_type>(); //内存池分配内存

        c.upstream.connect_timeout = 1000*30;    //设置超时时间, 重要
        c.upstream.send_timeout    = 1000*30;
        c.upstream.read_timeout    = 1000*30;
        c.upstream.buffer_size     = ngx_pagesize; //设置缓冲区大小, 重要

        return &c;
    }
};

```

注意在 `create()` 函数里我们必须设置超时时间和缓冲区大小这四个基本参数, 保证在后面启动 upstream 时正确连接。

9.6.3 业务逻辑类

`ndg_upstream_module` 模块的业务逻辑可以分解为两个类: `NdgUpstreamCallback` 实现回调函数, `NdgUpstreamHandler` 组装回调函数并启动 upstream 框架, 这样做可以避免逻辑代码太过复杂导致类过于庞大。

生成 upstream 请求

`create_request()` 函数的实现较为简单, 转发到上游的请求行直接使用 `$args`:

```

class NdgUpstreamCallback final //upstream 所需的回调函数
{
public:
    static ngx_int_t create_request(ngx_http_request_t* r)
    {
        ngx_str_t msg = ngx_string("xxxx"); //默认的请求头

        if(r->args.len >= 4) //检查请求参数
        {
            msg = r->args; //长度足够则使用$args
            msg.len = 4; //请求最多 4 个字节
        }

        NgxPool pool(r); //内存池
        NgxUpstream u(r); //upstream 代理对象
    }
};

```

```

    ngx_buf_t buf = pool.buffer();           //创建缓冲区
    buf.range(msg);                          //设置缓冲区数据

    ngx_chain_node_t ch = pool.chain();      //创建数据链表
    ch.set(buf);                             //设置缓冲区数据

    u.request(ch);                           //添加 upstream 请求数据

    return NGX_OK;
}

```

重新初始化请求

由于我们的 upstream 模块功能比较简单,所以 `reinit_request()` 函数没有特别的操作,是一个空函数。

处理响应头

真实的 upstream 模块处理响应头的代码非常复杂,通常需要使用状态机,并在 `ctx` 里保存解析的状态,但本节的后端服务器响应头只有两个字节,所以解析起来很容易:

```

public:
    static ngx_int_t process_header(ngx_http_request_t* r)
    {
        ngx_upstream_t u(r);                //代理 upstream 对象
        ngx_buf_t buf = u.buffer();          //检查从上游收到的数据

        if(buf.size() < 2)                   //如果长度不足
        { return NGX_AGAIN; }                //返回 AGAIN 要求继续接收

        auto p = buf->pos;                    //检查缓冲区里收到的头

        u.headers().content_length_n = (p[0] << 8) + p[1]; //设置响应数据长度

        u.headers().status_n = NGX_HTTP_OK;  //设置响应状态
        u.state().status = NGX_HTTP_OK;      //设置 upstream 状态

        buf.consume(2);                       //消费缓冲区里的两个字节

        return NGX_OK;                       //已经处理完响应头
    }
};                                           //回调函数类结束

```

`process_header()` 函数必须把上游服务器的响应状态转换成下游能够接收的形式(也

就是 HTTP 格式), 解析出响应数据的状态码、数据长度等基本参数, 最后“消费”掉接收的头数据。

读者可以进一步参考 `ngx_http_memcached_module` 和 `ngx_http_proxy_module` 等模块的源码, 了解响应头的解析处理逻辑。

启动 upstream 机制

类 `NdgUpstreamHandler` 使用 `NdgUpstreamCallback` 实例化了 `NgxUpstreamHelper`, 在处理函数中初始化 upstream, 设置连接参数, 然后启动 upstream 机制。

由于 upstream 机制相关的代码均已经封装在了 `NgxUpstreamHelper` 里, 所以处理函数的代码很简洁:

```
class NdgUpstreamHandler final                                //业务逻辑类
{
public:
    typedef NdgUpstreamHandler    this_type;                //简化类型定义
    typedef NdgUpstreamCallback    callback_type;

    typedef NdgUpstreamModule        this_module;            //本模块类型定义
    typedef NgxUpstreamHelper<callback_type::create_request,
                             &callback_type::reinit_request,
                             &callback_type::process_header>
                             this_upstream;                //回调函数实例化

public:
    static ngx_int_t handler(ngx_http_request_t *r)
    {
        NgxRequest req(r);
        if(!req.method(NGX_HTTP_GET))                        //处理原请求
                                                                //检查请求方法
        {
            return NGX_HTTP_NOT_ALLOWED;
        }

        this_upstream u(r);                                  //初始化 upstream

        auto& cf = this_module::conf().loc(r);                //获取模块配置

        u.conf(cf.upstream);                                  //设置 upstream 参数

        return u.start();                                      //启动 upstream 框架
    }
};
```

在实例化 `NgxUpstreamHelper` 时我们只给出了三个模板参数，第四个模板参数 `finalize_request` 使用了默认的空函数，节约了代码。

9.6.4 模块集成类

`NdgUpstreamInit` 整合模块的配置信息类和业务逻辑类，对于 `upstream` 模块来说重点是配置指令的解析处理，所以本节省略了函数指针表和模块定义。

类定义

`NdgUpstreamInit` 的定义如下：

```
class NdgUpstreamInit final
{
public:
    typedef NdgUpstreamConf      conf_type;        //简化类型定义
    typedef NdgUpstreamHandler   handler_type;
    typedef NdgUpstreamInit      this_type;

public:
    ...
    //其他成员函数，见后
};
```

配置指令解析

`upstream` 模块必须在指令解析函数里向 `ngx_http_core_module` 注册处理函数，并且调用 `ngx_http_upstream_add()`，设置使用的 `upstream` 服务器：

```
private:
    static char* set_upstream_pass(ngx_conf_t* cf,
                                   ngx_command_t* cmd, void* conf)
    {
        ngx_url_t u;                                //url 对象

        u.url = NgxStrArray(cf->args)[1];           //得到配置字符串
        u.no_resolve = true;

        auto& lcf = *reinterpret_cast<conf_type*>(conf); //当前的配置结构体

        lcf.upstream.upstream =                      //设置 upstream 服务器
            ngx_http_upstream_add(cf, &u, 0);

        if(!lcf.upstream.upstream)
        { return NGX_CONF_ERROR; }
```

```

NgxHttpCoreModule::handler(                //注册处理函数
    cf, &handler_type::handler);

    return NGX_CONF_OK;
}

```

配置指令数组是:

```

public:
    static ngx_command_t* cmds()
    {
        static ngx_command_t n[] =
        {
            {
                ngx_string("ndg_upstream_pass"), //配置指令名
                NgxTake(NGX_HTTP_LOC_CONF, 1),   //参数数量设置
                &this_type::set_upstream_pass,     //指令解析函数
                NGX_HTTP_LOC_CONF_OFFSET,
                0, nullptr
            },                                     //其他参数使用默认值

            ngx_null_command                       //空指令结束数组
        };

        return n;
    }

```

9.6.5 实现源文件

ndg_upstream_module 的实现源文件仍然是两行代码:

```

#include "NdgUpstreamInit.hpp "                //包含头文件
auto ndg_upstream_module = NdgUpstreamInit::module(); //ngx_module_t 变量

```

9.6.6 编译脚本

upstream 模块属于 http 模块, 在编译脚本里要使用类型 HTTP:

```

ngx_addon_name=ndg_upstream_module
ngx_module_type=HTTP                    #注意模块类型是 HTTP
ngx_module_name=ndg_upstream_module
ngx_module_srcs="$ngx_addon_dir/ModNdgUpstream.cpp"

```

. auto/module

静态模块的编译命令是:

```
./configure --add-module=path/to/upstream;make
```

9.6.7 测试验证

因为 upstream 模块要访问外部的服务器，所以我们在 Nginx 的配置文件中先要在 upstream 块里定义上游服务器列表：

```
upstream backend{                                #定义上游服务器列表
    server 127.0.0.1:2017;                        #后端服务位于本机 2017 端口
}
```

然后定义一个新的 location，在里面使用 ndg_upstream_pass 转发请求：

```
location /upstream {                             #定义 location
    ndg_upstream_pass backend;                    #转发请求到 backend 的服务器
}
```

使用 curl 发送请求，ndg_upstream_module 会把请求转发给本机的 2017 端口：

```
curl http://localhost/upstream -v                #请求会转发到 localhost:2017 处理
```

得到的结果可能是：

```
xxxxx1484207079                                #从 2017 端口得到的数据
```

9.7 开发 load-balance 模块

load-balance 模块的开发相对来说比较简单，回调函数只有四个，而且很多操作步骤都是模式化的，模块的重点是用算法操纵 IP 地址列表，选择出合适的上游服务器地址。

9.7.1 模块设计

本节的模块使用随机数作为负载均衡算法，随机选择 IP 地址，设计如下：

- 模块名是 ndg_load_balance_module;
- 配置指令是 ndg_load_balance;
- 不支持 weight/down 等附加配置参数;
- 使用 boost.random 库产生随机数。

9.7.2 配置信息类

ndg_load_balance_module 没有任何配置数据需要存储，所以无须定义配置信息类，

也就不需要模块单件类来访问配置数据或者请求环境数据。

当然 load-balance 模块也可以和其他 http 模块一样使用配置文件,但要用 NGX_HTTP_SRV_CONF_OFFSET 指定存储位置。这是因为 upstream 块没有 location 层次,使用 NGX_HTTP_LOC_CONF_OFFSET 没有意义。

9.7.3 业务逻辑类

与开发 upstream 模块类似,load-balance 模块最好也把回调函数放在一个单独的类里实现,再用 handler 类组装起来。

算法数据结构

类 NdgLoadBalanceCallback 把算法数据结构和算法捆绑在了一起,不需要再定义额外的数据结构。

注意在声明数据成员时一定要把 ngx_http_upstream_rr_peer_data_t 作为第一个成员,这样在调用 ngx_http_upstream_init_round_robin_peer() 函数时才能正确设置 peer.data。

NdgLoadBalanceCallback 的定义如下:

```
class NdgLoadBalanceCallback final
{
public:
    typedef NgxLogDebug          log;           //记录运行日志
    typedef NdgLoadBalanceCallback this_type;   //简化类型定义
    typedef boost::rand48        random_type;   //随机数发生器
public:
    ngx_http_upstream_rr_peer_data_t rrp;       //必须是第一个成员!

    u_char tries = 0;                          //重试次数
    ngx_event_get_peer_pt get_rr_peer =        //默认的 round robin 算法
        ngx_http_upstream_get_round_robin_peer;

public:
    ...                                         //其他成员函数,见后
};
```

算法实现

get() 函数从 IP 地址列表里随机选择一个,然后设置 ngx_peer_connection_t 对象的 socket 连接参数:

```

public:
    static ngx_int_t get(ngx_peer_connection_t *pc, void *data)
    {
        auto& peer_data = *reinterpret_cast<this_type*>(data);
        auto& peers = *peer_data.rrp.peers;           //IP 地址列表

        if(peer_data.tries++ > 5 || peers.single)     //检查重试次数
        {
            return peer_data.get_rr_peer(pc, &peer_data.rrp); //退化为 round robin
        }

        static random_type rand(time(0));            //随机数发生器

        auto& peer = peers.peer[rand() % peers.number]; //随机选择一个地址

        pc->sockaddr      = peer.sockaddr;           //设置连接参数
        pc->socklen        = peer.socklen;
        pc->name           = &peer.name;
        pc->cached         = false;
        pc->connection     = nullptr;

        log(pc->log).print("peer=%V", &peer.name);   //记录运行日志

        return NGX_OK;                               //成功得到了服务器地址
    }

```

初始化算法

类 NdgLoadBalanceHandler 使用回调函数类实例化 NgxLoadBalance:

```

class NdgLoadBalanceHandler final
{
public:
    typedef NdgLoadBalanceHandler    this_type;      //简化类型定义
    typedef NdgLoadBalanceCallback   callback_type;
    typedef NdgLoadBalanceCallback   peer_data_type;

    typedef NgxLoadBalance<peer_data_type,           //实例化 NgxLoadBalance
                        &callback_type::get>         //只有 get 回调
                        this_load_balance;

public:
    ...                                              //其他成员函数，见后
};

```

实例化后的 NgxLoadBalance 已经封装好了基本的初始化逻辑，如果不设置算法数据，直接调用就可以完成模块的初始化工作：

Nginx 完全开发指南：使用 C、C++和 OpenResty

```

public:
    static ngx_int_t init(ngx_conf_t *cf, ngx_http_upstream_srv_conf_t *us)
    {
        this_load_balance::init(cf, us, &this_type::init_peer);
        return NGX_OK;
    }
public:
    static ngx_int_t init_peer(ngx_http_request_t *r,
                                ngx_http_upstream_srv_conf_t *us)
    {
        this_load_balance::init(r, us); //忽略返回的数据对象
        return NGX_OK;
    }
};

```

9.7.4 模块集成类

因为没有配置信息类，所以 `NdgLoadBalanceInit` 只需使用业务逻辑类来初始化 load-balance 模块入口。

类定义

`NdgLoadBalanceInit` 的定义如下：

```

class NdgLoadBalanceInit final
{
public:
    typedef NdgLoadBalanceHandler    handler_type; //简化类型定义
    typedef NdgLoadBalanceInit      this_type;
public:
    ... //其他成员函数，见后
};

```

配置指令解析

load-balance 模块需要在指令解析函数里初始化入口函数，可以直接调用 `NgxHttpUpstreamModule::init()` 实现：

```

private:
    static char* set_load_balance(ngx_conf_t* cf,
                                   ngx_command_t* cmd, void* conf)
    {
        NgxHttpUpstreamModule::init(cf, &handler_type::init);
        return NGX_CONF_OK;
    }

```

指令数组里需要注意的是使用参数 `NGX_HTTP_UPS_CONF`，而且参数数量是 0：

```
public:
    static ngx_command_t* cmds()
    {
        static ngx_command_t n[] =
        {
            {
                ngx_string("ndg_load_balance"), //配置指令名
                NgxTake(NGX_HTTP_UPS_CONF, 0), //注意这里
                &this_type::set_load_balance, //指令解析函数
                0, 0, nullptr
            },

            ngx_null_command //空指令结束数组
        };

        return n;
    }
```

函数指针表

`ndg_load_balance_module` 不需要配置任何参数，所以在配置解析阶段不需要任何回调函数，8 个函数指针都是空：

```
public:
    static ngx_http_module_t* ctx()
    {
        static ngx_http_module_t c =
        {
            NGX_MODULE_NULL(8) //宏生成 8 个空指针
        };

        return &c;
    }
```

9.7.5 实现源文件

`ndg_upstream_module` 的实现源文件如下：

```
#include "NdgLoadBalanceInit.hpp" //包含头文件
auto ndg_load_balance_module = NdgLoadBalanceInit::module();
```

9.7.6 编译脚本

load-balance 模块虽然不处理请求，但它仍然是 http 模块，所以在编译脚本里也要使用类型 HTTP：

```
ngx_addon_name=ndg_load_balance_module
ngx_module_type=HTTP #注意模块类型是 HTTP
ngx_module_name=ndg_load_balance_module
ngx_module_srcs="$ngx_addon_dir/ModNdgLoadBalance.cpp"

. auto/module
```

静态模块的编译命令是：

```
./configure --add-module=path/to/loadbalance;make
```

9.7.7 测试验证

可以为 9.6 节的 ndg_upstream_module 模块增加一些上游服务器的配置：

```
upstream backend{
    ndg_load_balance; #定义上游服务器列表
    server 127.0.0.1:2017; #启用负载均衡算法
    server localhost:2018; #本机 2017 端口服务器
    #本机 2018 端口服务器
}
```

仍然使用 curl 发送请求，ndg_load_balance_module 会随机选择一个虚拟主机处理请求，可以通过运行日志看到运行结果，例如：

```
2017/xx/xx [error]: peer=127.0.0.1:2018 while connecting to upstream ...
```

9.8 总结

本章讲解了 Nginx 里两种特殊的 http 模块：upstream 模块和 load-balance 模块，它们都工作在 upstream 框架里，是 Nginx 宫殿里的阁楼和瞭望塔。

模块 ngx_http_upstream_module 实现了 upstream 框架，它可以高效地与上游服务器通信，转发下游的请求，再回传上游服务器返回的响应数据，支持 HTTP、Memcached、Redis 等多种协议，让 Nginx 有能力承担起核心节点的重任，分类或整合 Web 应用。

upstream 框架是 Nginx HTTP 框架的一部分，本质上是 content handler 模块。它在内容产生流程的关键节点上定义了若干回调函数，执行选择上游地址、组织请求头、解析响

应数据等操作，其他模块需要实现这些回调函数才能利用 upstream 框架完成整个请求转发工作。

由于 upstream 框架实现了主要的请求转发逻辑，所以开发 upstream 模块和 load-balance 模块都较为简单，只需要实现少量回调函数就可以。但它们的开发步骤略微烦琐，需要在配置解析阶段做的事情比较多，必须按照框架的要求，设置好各种参数和回调函数。

upstream 模块必须实现的是 `create_request`、`reinit_request`、`process_header` 和 `finalize_request` 四个回调函数，然后在处理函数里设置好连接上游的参数，调用 `ngx_http_upstream_init()` 就可以启动 upstream 机制收发数据。

load-balance 模块的核心是实现算法的 `get/free` 函数，但要想把模块接入 upstream 框架需要在配置解析阶段做很多准备，例如设置模块入口、初始化 IP 地址列表等。

本章实现了四个 C++ 类，在模板参数列表里确定了回调函数，较好地封装了 upstream 模块和 load-balance 模块的初始化和设置操作，有助于我们的模块开发工作。

第 10 章

Nginx HTTP子请求

Nginx HTTP 框架定义了四种模块：handler、filter、upstream 和 load-balance，这四种模块互相协作可以处理绝大多数 HTTP 请求，但它们还存在一定的局限性：只能处理单一的请求，只能完成一件任务，对于要执行多个请求才能完成的复杂任务无能为力。例如，向数个不同的上游服务器获取数据，最后整理成一个汇总表格输出，显然任何一种模块都无法胜任。

如果是在客户端，这不是个问题，客户可以自己分解任务，逐个向 Nginx 发送请求，得到数据后再自行整理。这种方法虽然可行却增加了客户端的负担，而且运行效率得不到保证。

Nginx 为此设计了子请求机制，把业务处理逻辑由客户端转移到了服务器端，目的是分解复杂的处理逻辑，把操作流程拆分成若干个相对简单的“子请求”。每一个子请求都可以由现有的 handler 或 upstream 模块完成，并且这些子请求仍然可以享用 Nginx 框架的异步高性能的好处，不会有效率上的损失，最后由主请求整合数据返回给客户端。

本章将详细介绍 Nginx 的子请求机制，使用它可以实现复杂的业务逻辑，让 Nginx 变身为一台高效的应用服务器。

10.1 子请求简介

由客户端发起的 HTTP 请求我们称之为主请求，它直接与客户端进行 TCP 通信，解析完请求头和请求体后在处理引擎中经过完整的 11 个阶段，最终产生响应数据输出，是典型的 RPC (remote procedure call)。

子请求是由 Nginx 内部发起的特殊 HTTP 请求，因为已经在服务器内部，所以它不需要建立 TCP 连接，也没有请求数据解析的成本，直接找到对应的 location 进入处理引擎处理，

相对于主请求资源消耗较少，本质上与发起一个函数调用没有区别，是 LPC (local procedure call)。

子请求与主请求的功能和处理流程基本相同，都是由 Nginx 的 HTTP 处理引擎处理，使用相同的 handler 模块和 filter 模块链，区别主要在于两者的发起方式。子请求是 Nginx 内部的 LPC，所以没有 TCP 连接和请求解析的成本，但因为不直接与客户端通信，所以它不能直接发送数据，通常也不会记录访问日志。^①

子请求也存在功能限制，它不能跨 server 调用（因为不发起 TCP 连接），只能调用本 server 块内的 location。不过我们可以利用 upstream 机制，在 location 里使用 proxy_pass 等反向代理指令访问其他服务器，再用子请求调用 location，从而间接达到目的。

这里我们使用了“调用”这个词，它非常精确地描述了子请求的作用：可以把 Nginx 配置文件里的 location 当作是功能函数，通过子请求向它传递 uri、headers、body 等参数，“调用”后得到结果数据。

子请求的出现，让 location 不再是孤立的个体，而是成为了处理业务的有机组件，使用 location 完成多个简单的请求任务，再用子请求调用它们，最终即可实现复杂的应用逻辑。

10.1.1 工作原理

主请求可以发起多个子请求，而且子请求还可以再发起子请求，形成树状的调用层次。灵活使用子请求可以在 Nginx 服务器里访问任意的资源，最后由主请求产生响应数据返回。

由于存在多级的请求树，所以发起子请求的请求又称为父请求，它的概念与主请求不同。主请求是绝对概念，只能有一个，相当于树的根，而父请求是相对的概念，是发起当前请求的上一级请求。主请求一定是父请求，而父请求则不一定是主请求。

发起子请求处理的基本流程是：

- 1) 客户端发起请求，即主请求，由 Nginx 处理；
- 2) 主请求调用 ngx_http_subrequest() 创建出子请求对象，加入待处理请求链表；
- 3) 主请求返回 NGX_DONE，暂停主请求的执行，等待子请求完成；

^① 如果要让子请求记录访问日志需要使用配置指令 log_subrequest on。

- 4) HTTP 处理引擎从待处理请求链表里获取排队的子请求，依次处理；
- 5) 子请求的处理过程与主请求完全相同，还可以再发起子请求（即 2~4 步）；
- 6) 子请求处理完毕，使用回调函数通知主请求继续运行；
- 7) 主请求获取子请求生成的响应数据，加工后返回给客户端。

子请求的创建和处理流程可以用图 10-1 来表示。

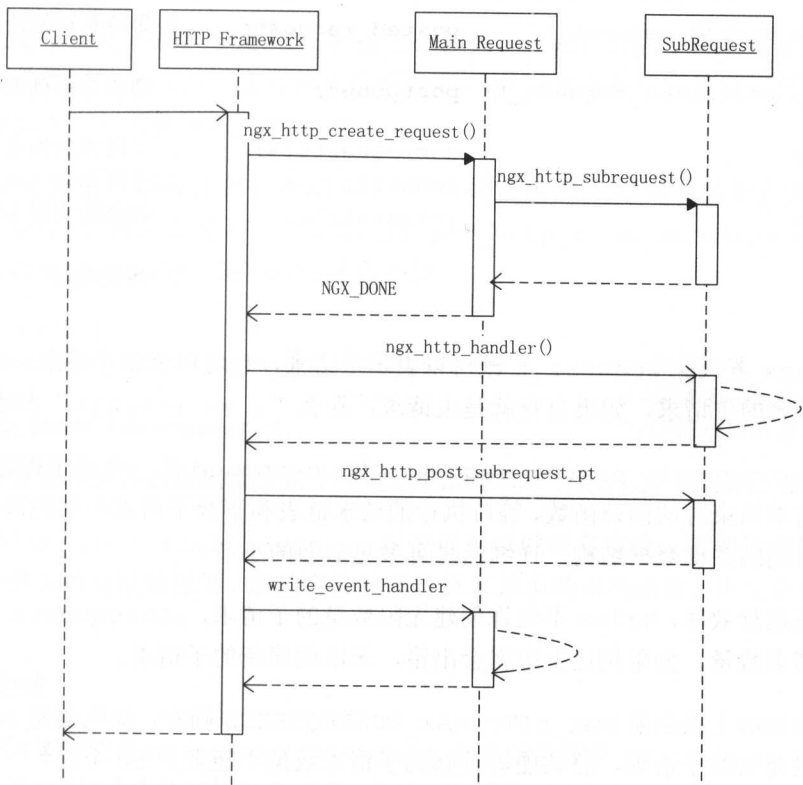


图 10-1 子请求的创建和处理流程

10.1.2 请求结构体

子请求与主请求只有从属关系，本质上没有区别，所以结构体 ngx_http_request_t 也用来表示子请求对象。

ngx_http_request_t 里子请求相关的成员如下：

```
// 定义在 http/nginx_http_request.h
struct ngx_http_request_s {

    ngx_http_event_handler_pt    read_event_handler;    //读事件回调函数
    ngx_http_event_handler_pt    write_event_handler;    //写事件回调函数

    ngx_http_request_t*          main;                    //请求所属的主请求
    ngx_http_request_t*          parent;                  //请求所属的父请求

    ngx_http_post_subrequest_t*   post_subrequest;        //子请求回调函数对象

    ngx_http_posted_request_t*    posted_requests;        //等待执行的请求链表

    ngx_http_postponed_request_t* postponed;              //请求数据链表

    unsigned                      subrequests:8;          //子请求的数量
    unsigned                      count:8;                //引用计数
    unsigned                      internal:1;              //内部请求标志位

    ...                                                    //其他成员
};
```

结构体 `ngx_http_request_t` 既可以表示主请求，也可以表示子请求。成员 `main` 指向当前请求所属的主请求，如果自身就是主请求，那么“`r == r->main`”表达式为真。

`post_subrequest`、`posted_requests` 和 `postponed` 是三个名字相近的成员^①，它们保存了子请求结束时的回调函数、等待执行的请求链表和下级子请求产生的数据，是 Nginx 子请求运行机制的关键数据结构，详细信息可参见后面的小节。

为了保证运行效率，Nginx 不允许创建无限数量的子请求，`subrequests` 成员用来记录主请求的子请求数量，如果超过上限就会出错，无法创建新的子请求。

子请求数量的上限由宏 `NGX_HTTP_MAX_SUBREQUESTS` 确定，默认值是 50，也就是说，一个主请求以及它的子请求，总共能够发起的子请求数量不能多于 50 个。^②

10.1.3 回调函数

子请求的处理过程是异步的，创建子请求后，当前的请求（即父请求）必须返回 `NGX_DONE` 暂时中断处理，HTTP 处理引擎会去执行新发起的子请求。

① 与第 9 章负载均衡模块里的 `round robin` 算法情形比较相似。

② Nginx 1.10.0 之前的子请求最大数量是 200 个。

父请求需要设置子请求的回调函数，而子请求处理结束时必须要用回调函数通知父请求，恢复父请求的运行。

子请求回调函数

当子请求执行完毕后（成功处理或者发生错误），Nginx 会调用函数 `ngx_http_post_subrequest_pt`，它的声明是：

```
typedef ngx_int_t (*ngx_http_post_subrequest_pt) (
    ngx_http_request_t *r, void *data, ngx_int_t rc);
```

`ngx_http_post_subrequest_pt` 有三个参数，`r` 是当前的子请求对象（注意不是父请求）；`data` 的类型是 `void*`，允许传入任意数据；`rc` 是请求执行完毕后的错误码，如果执行成功通常是 `NGX_OK` 或者 `NGX_HTTP_OK`。

因为子请求的执行是异步的，所以 `data` 参数很重要，相当于函数的运行环境数据（context）。Nginx 特意定义了一个数据结构 `ngx_http_post_subrequest_t`，结构体里的 `data` 成员就是 handler 调用时传递的参数：

```
typedef struct {
    ngx_http_post_subrequest_pt handler;           //回调函数
    void* data;                                    //函数的相关数据
} ngx_http_post_subrequest_t;                    //子请求回调函数对象
```

它把回调函数和所需的数据绑定在了一起，用法相当于 C++ 里的 lambda 表达式（闭包）。

在 `ngx_http_post_subrequest_pt` 里我们可以做很多事情，例如检查处理状态，检查子请求输出的头和响应数据等，但最重要的是设置父请求的回调函数，让父请求可以继续运行。

父请求回调函数

父请求的回调函数是 `ngx_http_request_t` 里的函数指针 `write_event_handler`，即父请求被重新“激活”处理后执行的函数，也可以说是断点恢复后的后续处理流程，声明是：

```
typedef void (*ngx_http_event_handler_pt) (ngx_http_request_t *r);
```

它与 7.2.1 节的 HTTP 处理函数 `ngx_http_handler_pt` 很像，只是返回值类型不同。

在不同的处理阶段，父请求的回调函数也不相同。

对于 `rewrite`、`access` 等非内容产生阶段，由于整个处理流程还未走完，所以回调函数通常是 `ngx_http_core_run_phases()`，让父请求继续在引擎里处理。

而在内容产生阶段 (NGX_HTTP_CONTENT_PHASE), 由于处理引擎已经执行完毕, 所以回调函数应该是父请求自己的处理逻辑, 整合从子请求收到的数据, 发送数据, 最后必须调用 `ngx_http_finalize_request()` 来结束请求。

10.1.4 待处理请求链表

结构体 `ngx_http_request_t` 里的 `posted_requests` 成员以链表的方式管理主请求发起的所有子请求, 由 HTTP 处理引擎调度执行。注意: 它只在主请求里有意义。

主请求可以发起多个子请求, 这些子请求是平等的关系, 但存在先后顺序, 所以 Nginx 使用一个链表结构串起当前请求的所有子请求, 链表的定义是:

```
typedef struct ngx_http_posted_request_s  ngx_http_posted_request_t;

struct ngx_http_posted_request_s {
    ngx_http_request_t*      request;          //子请求对象
    ngx_http_posted_request_t* next;          //下一个链表节点
};
```

可以看到, `ngx_http_posted_request_t` 就是一个非常简单的链表, 使用指针来保存子请求对象。

在函数 `ngx_http_run_posted_requests()` 里 Nginx 会获取主请求的待处理请求链表, 逐个执行请求的 `write_event_handler` 回调函数, 驱动请求的处理流程。

10.1.5 子请求存储结构

虽然子请求的发起有先后顺序, 但它们的完成顺序是不固定的, 由于任务的不同, 很可能后发起的子请求会先完成。为了能够正确组织子请求返回的数据, Nginx 使用 `ngx_http_request_t::postponed` 组织本级请求发起的所有下级子请求。

`postponed` 的类型是 `ngx_http_postponed_request_t`, 定义如下:

```
typedef struct ngx_http_postponed_request_s  ngx_http_postponed_request_t;

struct ngx_http_postponed_request_s {
    ngx_http_request_t*      request;          //子请求对象
    ngx_chain_t*              out;              //子请求产生的数据
    ngx_http_postponed_request_t* next;        //下一个链表节点
};
```

它主要用于 `ngx_http_postpone_filter_module`，整合子请求对象和子请求产生的数据。

10.2 子请求运行机制

子请求的运行机制比 `handler/filter/upstream` 等模块的运行机制要复杂一些，不仅有多个请求，而且这些请求之间还会彼此通信，协同工作。处理逻辑也分散在 Nginx 框架的各个角落，想要理解它的运行机制必须要多花些力气。

Nginx 处理子请求有三个基本步骤：创建子请求、调度子请求/父请求和组织整理所有请求产生的数据。

10.2.1 创建子请求

在任意的 HTTP 处理阶段都可以调用函数 `ngx_http_subrequest()` 创建子请求对象。

函数声明

`ngx_http_subrequest()` 的接口参数较多，是 Nginx 里比较少见的，声明如下：

```
ngx_int_t ngx_http_subrequest(ngx_http_request_t *r,  
    ngx_str_t *uri, ngx_str_t *args, ngx_http_request_t **sr,  
    ngx_http_post_subrequest_t *psr, ngx_uint_t flags);
```

入口参数的含义是：

- `r` : 当前的请求对象，即父请求；
- `uri` : 子请求的 uri，也就是本 server 块内的某个 location 的名字；
- `args` : 子请求的 uri 参数，可以为空 (`nullptr`)；
- `sr` : 输出参数，传出创建好的子请求对象；
- `psr` : 子请求结束时的回调函数，见 10.1.3 节；
- `flags` : 标志位，定制子请求的某些行为，通常是 0。

创建一个子请求必须要确定的是 `uri` 参数，它决定从哪个 location 里执行子请求获得数据。`args` 参数是可选的，如果 location 支持参数，就可以传递 uri 之外的附加信息。

`flags` 可以取值为 `NGX_HTTP_SUBREQUEST_IN_MEMORY`，会置子请求结构体里的 `subrequest_in_memory` 标志位为 `true`，这样 `upstream` 模块获取的上游数据都会保存在

内存里，方便后续处理，对于非 `upstream` 模块则没有意义。^①

函数实现

`ngx_http_subrequest()` 的内部实现对我们了解子请求的工作原理很重要，下面就详细分析它的代码（省略了些无关的部分）：

```
ngx_int_t
ngx_http_subrequest(ngx_http_request_t *r,
    ngx_str_t *uri, ngx_str_t *args, ngx_http_request_t **psr,
    ngx_http_post_subrequest_t *ps, ngx_uint_t flags)
{
    if (r->subrequests == 0) {                //子请求达到上限
        return NGX_ERROR;                     //子请求创建失败，返回错误
    }
}
```

函数首先检查子请求计数器，如果超过了 `NGX_HTTP_MAX_SUBREQUESTS`（当前是 50）的限制，那么就不能再创建新的子请求，返回 `NGX_ERROR` 错误码。

接下来的代码是创建子请求对象，与创建请求对象的 `ngx_http_create_request()` 类似，只是大部分字段都是直接拷贝自父请求，无须重新分配内存：^②

```
sr = ngx_pcalloc(...);                      //创建子请求对象
sr->signature = NGX_HTTP_MODULE;             //结构体的签名字符串

sr->ctx = ngx_pcalloc(...);                  //创建模块的 ctx 存储空间

cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
sr->main_conf = cscf->ctx->main_conf;         //模块的各层次配置数据
sr->srv_conf = cscf->ctx->srv_conf;
sr->loc_conf = cscf->ctx->loc_conf;

sr->pool = r->pool;                           //复用父请求的内存池
sr->headers_in = r->headers_in;               //复用父请求的请求头
sr->request_body = r->request_body;           //复用父请求的请求体

sr->method = NGX_HTTP_GET;                    //请求方法默认为 GET
```

① 另一个取值是 `NGX_HTTP_SUBREQUEST_WAITED`，但目前在 Nginx 里并没有太大用处。

② 子请求的创建有个隐患：只拷贝了头节点（`headers_in.headers`），但内部的链表指针却没有同步调整，如果子请求改写头就可能造成父请求处理错误。解决办法是完全深拷贝 `headers` 结构体，让子请求与父请求的头信息链表完全独立。但一般来说子请求不会去改写头，所以问题不大。也许是出于这个考虑（还有效率），Nginx 一直没有对此做出修改。

```

sr->http_version = r->http_version;           //拷贝 http 版本信息

sr->request_line = r->request_line;             //复用原请求行
sr->uri = *uri;                                 //改写子请求的 uri

if (args) {                                    //改写子请求的 args
    sr->args = *args;
}

//设置子请求的标志位
sr->subrequest_in_memory = (flags & NGX_HTTP_SUBREQUEST_IN_MEMORY) != 0;
sr->waited = (flags & NGX_HTTP_SUBREQUEST_WAITED) != 0;

sr->unparsed_uri = r->unparsed_uri;             //复用原始 uri
sr->method_name = ngx_http_core_get_method;     //请求方法字符串默认为 GET
sr->http_protocol = r->http_protocol;           //复用原请求协议

ngx_http_set_exten(sr);                        //改写子请求的扩展名

```

上面的一大段代码都是设置子请求 HTTP 相关的各种参数，例如 method、uri、args 等。值得注意的是子请求的内存池与父请求的内存池是同一个，这意味着在子请求里分配的内存是一直可用的，不会因为子请求的销毁而消失^①，而请求的环境数据 ctx 是重新分配的空间，所以子请求和父请求的模块 ctx 数据是彼此独立的。

子请求的默认方法是 GET，而且其他参数也基本复制了父请求，但这并不是说子请求只能用 GET 方法，或者只能原样转发父请求的所有参数。在 ngx_http_subrequest() 执行后，我们可以使用输出参数 sr 获得创建好的子请求对象，它只是一个请求的“样板”，完全可以对它做任意的修改，例如把 method 改成 POST、再次修改 args、增加新的请求体数据，对此 Nginx 没有限制，用户可以非常灵活地定制子请求。

函数的后半部分是子请求调度相关数据的设置，重点是设置子请求的处理函数为 ngx_http_handler()：

```

sr->main = r->main;                             //设置主请求
sr->parent = r;                                  //设置父请求
sr->post_subrequest = ps;                        //子请求结束时的回调函数对象

sr->write_event_handler = ngx_http_handler;     //子请求的执行函数

pr = ngx_palloc(...);                          //子请求数据对象

```

① 这同时也意味着发起过多的子请求会造成内存不能及时释放，导致 Nginx 进程占用过多内存。

```

if (r->postponed) {
    for (p = r->postponed; p->next; p = p->next) //挂到父请求数据链表的末尾
        p->next = pr;

    } else {
        r->postponed = pr; //空链表则直接添加
    }

    sr->internal = 1; //标记是内部请求，即子请求
    sr->subrequests = r->subrequests - 1; //子请求计数器减 1

    tp = ngx_timeofday(); //计时器开始计时
    sr->start_sec = tp->sec;
    sr->start_msec = tp->msec;

    r->main->count++; //引用计数增加

    *psr = sr; //创建完毕，返回子请求对象

    return ngx_http_post_request(sr, NULL); //加入待执行请求链表
}

```

函数 `ngx_http_post_request()` 执行收尾工作，负责把子请求加入到主请求的待执行请求链表 `posted_requests` 里，这是子请求能够运行的重要步骤：

```

ngx_int_t
ngx_http_post_request(ngx_http_request_t *r, ngx_http_posted_request_t *pr)
{
    ngx_http_posted_request_t **p;

    if (pr == NULL) { //创建一个链表节点
        pr = ngx_palloc(r->pool, sizeof(ngx_http_posted_request_t));
    }

    pr->request = r; //设置链表里的请求对象
    pr->next = NULL;

    for (p = &r->main->posted_requests; *p; p = &(*p)->next) //找到链表末尾
        *p = pr; //添加到链表末尾等待调度

    return NGX_OK;
}

```

这样，在 `ngx_http_subrequest()` 执行完毕后，主请求里的执行请求链表 `posted_`

requests 就会增加一个新节点，但子请求并没有立即开始运行，而是等待处理引擎调度。

10.2.2 处理引擎

子请求的处理涉及 HTTP 框架里的多个函数，本节只简要介绍两个比较重要的函数。

子请求调度

`ngx_http_run_posted_requests()` 是驱动子请求运行的关键函数，代码摘要如下：

```
void
ngx_http_run_posted_requests(ngx_connection_t *c)
{
    for ( ;; ) {
        pr = r->main->posted_requests;           //获取待执行的请求链表

        r->main->posted_requests = pr->next;       //移除头节点
        r = pr->request;                          //得到当前待执行的请求

        r->write_event_handler(r);                //执行请求的 handler
    }
}
```

函数 `ngx_http_run_posted_requests()` 的逻辑很简单，遍历主请求的 `posted_requests` 成员，也就是待执行的请求链表，然后执行请求的 `write_event_handler` 回调函数。

因为在创建子请求时已经设置了 `write_event_handler` 为 `ngx_http_handler()`，所以它会把子请求的阶段设置到 `server_rewrite` (跳过了 `POST_READ` 阶段)，进而从头遍历引擎数组，让引擎里的 `http` 模块去处理子请求，流程与正常的请求相同。

`ngx_http_handler()` 里子请求的相关代码是：

```
//ngx_http_handler()函数，定义在http/ngx_http_core_module.c
r->phase_handler = cmcf->phase_engine.server_rewrite_index; //设置阶段起点
ngx_http_core_run_phases(r);                               //执行处理引擎
```

在处理完一个请求后，`ngx_http_run_posted_requests()` 会把请求从链表中移除，随着请求不断被处理，最终链表会全部清空。

结束子请求

请求或子请求结束时都会调用函数 `ngx_http_finalize_request()`，它是 Nginx HTTP 处理的核心函数之一，内部逻辑比较复杂（可参见 17.6 节），这里只介绍与子请求相关

的部分。

当子请求处理完毕正常结束时，`ngx_http_finalize_request()` 会调用 `post_subrequest` 成员，执行子请求结束时的回调函数，并把父请求加入待处理请求链表，等待引擎调度从而“激活”：

```
void ngx_http_finalize_request(ngx_http_request_t *r, ngx_int_t rc)
{
    ...
    if (r != r->main && r->post_subrequest) {           //判断子请求
        rc = r->post_subrequest->handler(                //执行子请求的回调函数
            r, r->post_subrequest->data, rc);             //传递请求的错误码
    }
    ...

    if (ngx_http_post_request(pr, NULL) != NGX_OK) { //父请求加入待处理链表
        ...
    }
}
```

因此，子请求需要在回调函数里设置好父请求的 `write_event_handler`，这样子请求结束后调度函数 `ngx_http_run_posted_requests()` 会在待处理请求链表里找到父请求，从而继续父请求的后续工作。

10.2.3 数据整理

现在父请求和子请求都可以正常运行了，但处理它们产生的数据又是一个新的问题，必须保证子请求产生的数据能够按照它们创建的先后顺序正确输出，而不能是异步执行完成后的乱序。

Nginx 为此实现了一个特殊的 filter 模块 `ngx_http_postpone_filter_module`，它专门处理子请求的数据，把数据插入到主请求 `postponed` 成员里的合适位置。

`ngx_http_postpone_filter_module` 是 Nginx 必须编译的内置模块，不能使用选项删除，是子请求运行机制里必不可少的组件。它相当于模块数据发送过程中的一个“钩子”，中断了子请求的过滤链表处理，把子请求产生的数据“钩”回到父请求。

`ngx_http_postpone_filter_module` 是 body filter，只处理响应体数据，不处理响应头，核心功能是函数 `ngx_http_postpone_filter()`，代码摘要如下：

```
static ngx_int_t
ngx_http_postpone_filter(ngx_http_request_t *r, ngx_chain_t *in)
```

```

{
    ngx_connection_t      *c;
    ngx_http_postponed_request_t  *pr;

    c = r->connection;                                //获得当前的连接对象

    if (r != c->data) {                                //r 不是当前正在处理的请求
        ngx_http_postpone_filter_add(r, in);           //数据加入父请求链表末尾
        return NGX_OK;                                 //直接结束过滤链表
    }

    if (r->postponed == NULL) {                         //没有需要延后发送的数据
        if (in || c->buffered) {                       //使用主请求发送数据
            return ngx_http_next_body_filter(r->main, in);
        }

        return NGX_OK;
    }

    if (in) {                                           //有需要延后发送的数据
        ngx_http_postpone_filter_add(r, in);           //当前数据加入父请求链表末尾
    }

    do {                                                //循环处理现有的待发送数据
        pr = r->postponed;                             //获取头节点

        if (pr->request) {                             //节点里是子请求

            r->postponed = pr->next;                   //移除头节点
            c->data = pr->request;                     //置为连接处理的当前请求
            return ngx_http_post_request(pr->request, NULL); //加入待处理请求链表
        }

        if (ngx_http_next_body_filter(                //节点是数据，由主请求发送
            r->main, pr->out) == NGX_ERROR) {
            return NGX_ERROR;
        }

        r->postponed = pr->next;                       //处理下一个节点
    } while (r->postponed);                             //直至所有的数据节点处理完毕

    return NGX_OK;
}

```

模块处理子请求的关键是 `c->data`，它表示当前处理的请求对象，也就是可发送数据的请求，是保证子请求数据正确顺序的关键，Nginx 通过调整 `c->data` 来控制子请求数据发送的顺序。

当响应数据流经模块时，如果请求的顺序不对，那么数据就会被加入主请求的 `postponed` 链表里，否则就会走主请求的过滤链表发送给客户端。

更细致地解析 `ngx_http_postpone_filter_module` 与 `postponed` 链表数据的关系需要较多的篇幅，而实际上它能发挥作用的时候并不是很多，故本书从略，读者只需理解它的基本工作原理就足够了，暂不必深究细节。

10.3 C++封装

在 Nginx 里提供的子请求的接口只有一个用于创建的 `ngx_http_subrequest()`，设置回调函数和子请求参数都要用户自己编写代码。本节实现两个简单的 C++ 类，分别封装回调函数和创建子请求。

10.3.1 NgxSubRequestHandler

通常使用子请求需要实现两个回调函数，所以我们使用类 `NgxSubRequestHandler` 把它们绑定在一起，增强内聚性。

模板参数

`NgxSubRequestHandler` 是一个模板类，它的第一个模板参数是客户需要实现的回调函数类，含有 `sub_post` 和 `parent_post` 两个回调函数，第二个模板参数是子请求所在的处理阶段。

回调函数类的形式是：

```
class SubrequestCallback //子请求处理所需的回调函数
{
public:
    static ngx_int_t sub_post(ngx_http_request_t* r, void* data, ngx_int_t rc);
    static ngx_int_t parent_post(ngx_http_request_t* r);
};
```

注意 `parent_post()` 函数的返回类型是 `ngx_int_t`，而不是 `void`，这样回调函数类可以专注于业务逻辑的实现，而 `NgxSubRequestHandler` 封装了子请求处理时必须做的设置父请求回调函数和调用 `ngx_http_finalize_request()`。

基本形式

在非内容产生阶段，父请求的回调函数应该是 `ngx_http_core_run_phases()`，以便让 HTTP 处理引擎继续执行后续的模块。

`NgxSubRequestHandler` 的定义如下：

```
template<typename T, ngx_http_phases ph>
class NgxSubRequestHandler final
{
public:
    typedef NgxSubRequestHandler this_type;
public:
    static ngx_int_t sub_post(ngx_http_request_t* r, void* data, ngx_int_t rc)
    {
        r->parent->write_event_handler =          //设置父请求的回调函数
            ngx_http_core_run_phases;
        return T::sub_post(r, data, rc);          //执行子请求的回调函数
    }
};
```

内容产生阶段特化

在内容产生阶段，必须明确指定父请求的回调函数，这样才能继续处理父请求，同时还要在父请求处理结束时调用 `ngx_http_finalize_request()`。`NgxSubRequestHandler` 使用模板技术，完全封装了这些流程化的操作：

```
template<typename T>                                //注意，模板特化
class NgxSubRequestHandler<T, NGX_HTTP_CONTENT_PHASE> final
{
public:
    typedef NgxSubRequestHandler this_type;
public:
    static ngx_int_t sub_post(ngx_http_request_t* r, void* data, ngx_int_t rc)
    {
        r->parent->write_event_handler =          //设置父请求的回调函数
            &this_type::parent_post;             //必须是自定义的回调函数
        return T::sub_post(r, data, rc);          //执行子请求的回调函数
    }

    static void parent_post(ngx_http_request_t* r) //父请求的回调函数
    {
        ngx_http_finalize_request(                //结束请求
            r, T::parent_post(r));                 //执行父请求的回调函数
    }
};
```

```
};
```

10.3.2 NgxSubRequest

类 `NgxSubRequest` 利用 `NgxSubRequestHandler` 包装子请求的回调函数，调用 `ngx_http_subrequest()` 创建出子请求对象，定义如下：

```
template<typename T, ngx_http_phases ph = NGX_HTTP_CONTENT_PHASE>
class NgxSubRequest final : public NgxWrapper<ngx_http_request_t>
{
public:
    typedef NgxWrapper<ngx_http_request_t>    super_type;    //简化类型定义
    typedef NgxSubRequest                    this_type;
    typedef NgxSubRequestHandler<T, ph>      this_handler;  //回调函数封装
public:
    NgxSubRequest(ngx_http_request_t* r):super_type(r)
    {}
    ~NgxSubRequest() = default;
public:
    ngx_http_request_t* create(ngx_str_t& uri,                //子请求 uri
                               ngx_str_t& args,              //子请求参数
                               void* psr_data = nullptr,     //回调使用的数据
                               ngx_uint_t flags = 0) const   //子请求标志位
    {
        auto psr =                                           //子请求回调函数
            NgxPool(get()).alloc<ngx_http_post_subrequest_t>();

        psr->handler = &this_handler::sub_post;             //设置回调函数
        psr->data = psr_data;                               //设置函数使用的数据

        ngx_http_request_t* sr;                             //子请求指针

        auto rc = ngx_http_subrequest(                      //创建子请求
            get(), &uri, &args, &sr, psr, flags);
        NgxException::require(rc);                           //检查错误码

        return sr;                                           //返回子请求对象
    }
};
```

这段代码里的关键是 `NgxSubRequestHandler` 的实例化，得到了子请求的回调函数，然后创建回调函数结构体，最后创建出子请求对象返回。

得到子请求对象后，用户可以修改它的 `method`、`uri`、`args` 等任意成员，定制出目标 location 所需的正确请求。

如果子请求创建失败, `create()` 会抛出异常, 客户代码需要使用 `try-catch` 块来捕获异常。

10.4 数据回传模块

在 Nginx 里创建子请求很容易, 但获取并处理子请求返回的数据却不是一件容易的事情。虽然 Nginx 提供了模块 `ngx_http_postpone_filter_module`, 但它处理的方式过于“死板”, 只能把数据拷贝到主请求的输出链里, 没有设置回调函数接口, 所以我们无法对数据做更多的加工处理, 限制了子请求的用途。

处理子请求数据的另一个办法是创建子请求时使用标志位 `NGX_HTTP_SUBREQUEST_IN_MEMORY`, 让 `upstream` 块把收到的数据都保存在 `r->upstream->buffer` 里, 这样在子请求结束的回调函数中就可以从 `buffer` 里获取数据, 然后回传给父请求。但这种方法有很大的局限性, 首先是它只能用于 `upstream` 模块, 不能用于普通的 `handler` 模块; 其次是数据必须保存在内存里, 而且受 `buffer_size` 大小的限制。所以, 本书不推荐使用 `NGX_HTTP_SUBREQUEST_IN_MEMORY` 标志处理子请求数据。

`ngx_http_postpone_filter_module` 功能虽然有限, 但为解决这个问题提供了很好思路。我们可以开发一个类似功能的 `filter` 模块, 它在 Nginx 过滤链表里安装一个“钩子”, 把子请求的响应头和响应体数据都“钩”回到父请求里, 让父请求能够自由使用任何 `http` 模块产生的数据。

10.4.1 模块设计

数据回传 `filter` 模块的基本设计如下:

- 模块名是 `ndg_data_hook_module`, 过滤处理响应头和响应体;
- 不使用配置指令, 没有配置数据结构;
- 在请求环境数据 (`ctx`) 里保存子请求的数据;
- 父请求需要在 `ctx` 里设置一个标志位, 子请求才会把数据传回父请求;
- 需要调整模块在 `ngx_modules` 数组中的位置, 必须在 `ngx_http_postpone_filter_module` 之前。

10.4.2 环境数据类

在 Nginx 里父请求和子请求的环境数据 `ctx` 的存储是互相独立的, 可以利用这一点在模块的 `ctx` 里存储我们想要的数 据, 通过 `r->parent` 得到父请求, 从而操作父请求里的模块

ctx。

类定义

ndg_data_hook_module 的环境数据类定义如下：

```
struct NdgDataHookCtx final
{
public:
    ngx_flag_t          hook = false;          //数据回传标志位，默认不处理

    ngx_int_t           status = 0;             //子请求的处理状态
    ngx_http_headers_out_t* headers = nullptr; //子请求的响应头数据
    ngx_chain_t*         body = nullptr;        //子请求的响应体数据

    ngx_flag_t          do_hook = false;        //子请求里处理 body 的标志位
public:
    ...                                         //其他成员函数，见后
};
```

NdgDataHookCtx 是本模块的核心数据结构，它使用 status、headers 和 body 保存子请求产生的数据，hook 和 do_hook 是两个标志变量，用来启用或者禁用数据回传功能。

启动数据回传

默认情况下 hook 变量值是 false，模块不会有任何动作，直接把数据转发到过滤链表的下一个节点，发起子请求后必须显式地调用成员函数 hooking() 启用数据回传功能：

```
public:
    void hooking()                                     //启用数据回传功能
    {
        hook      = true;                             //回传标志位置 true
        status     = 0;                                 //错误码初始化
        headers    = nullptr;                           //响应头初始化为空指针
        body       = nullptr;                           //响应体初始化为空指针
    }
```

在子请求里会检查父请求的模块 ctx，如果 hook==true，那么就需要把数据回传到父请求里。

设置处理状态

子请求的处理状态是一个重要的参数，父请求需要通过它来检查子请求执行是否成功，这只能在子请求结束时的回调函数里用参数 rc 来判断。NdgDataHookCtx 简单地封装了判断逻辑：


```

public:
void state(ngx_int_t rc) //检查子请求的处理状态
{
    if(status) //状态已经设置则不处理
    { return;}

    if(rc == NGX_OK) //错误码是 NGX_OK
    {
        status = NGX_HTTP_OK; //设置为 NGX_HTTP_OK
        return;
    }

    if(rc < NGX_OK) //发生 Nginx 内部错误
    {
        status = NGX_HTTP_INTERNAL_SERVER_ERROR; //设置为 500
        return;
    }

    status = rc; //设置为子请求的 HTTP 状态码
}

```

10.4.3 业务逻辑类

ndg_data_hook_module 是 filter 模块，需要利用 7.5.3 节的 NgxFiter 辅助类插入 Nginx 过滤链表，实现对响应数据的过滤处理。

类定义

NdgDataHookHandler 类定义如下：

```

class NdgDataHookHandler final //业务逻辑类
{
public:
    typedef NdgDataHookHandler this_type; //简化类型定义
    typedef NdgDataHookCtx ctx_type; //ctx 类型定义

    typedef NdgDataHookModule this_module; //模块定义
    typedef NgxFiter<this_type> this_filter; //过滤链表工具类定义

public:
    static ngx_int_t init(ngx_conf_t* cf) //postconfiguration 调用
    {
        this_filter::init( //插入过滤链表
            &this_type::header_filter, &this_type::body_filter);
        return NGX_OK;
    }
}

```

```
public:
    ... //其他成员函数，见后
};
```

处理响应头

当数据流经 `ndg_data_hook_module` 模块时，它需要检查父请求里是否设置了数据回传标志 `hook`，然后设置 `do_hook` 标志，告诉 `body filter` 要数据回传：

```
public:
    static ngx_int_t header_filter(ngx_http_request_t *r) //过滤响应头
    {
        auto pr = r->parent; //获取父请求
        if(!pr) //无父请求则不需要处理
        { return this_filter::next(r); }

        auto& ctx = this_module::ctx(); //准备获取模块的环境数据

        if(ctx.empty(pr)) //父请求未设置 ctx，不需要处理
        { return this_filter::next(r); }

        auto& pr_ctx_data = this_module::data(pr); //得到父请求的 ctx 数据

        if(!pr_ctx_data.hook) //父请求不要求回传数据
        { return this_filter::next(r); }

        this_module::data(r).do_hook = true; //父请求要求回传数据，设置标志

        pr_ctx_data.headers = &r->headers_out; //设置指针，回传响应头

        r->filter_need_in_memory = true; //子请求的参数设置
        r->header_sent = true;

        return NGX_OK; //结束过滤链表
    }
```

`header_filter()` 的代码比较多，因为它必须要检查父请求是否设置了 `hook` 标志。如果父请求要求 `hook`，那么它就设置 `do_hook`，用于之后的 `body_filter()`。

响应头回传的工作也可以在子请求结束回调时做，但在这里做更好，因为这只是一个指针的赋值，成本几乎为 0，而且可以节约回调函数里的客户代码。

处理响应体

`body_filter()` 的处理比较简单，只需要把数据挂到父模块的 `body` 指针上就可以了：

```

public:
    static ngx_int_t body_filter(ngx_http_request_t *r, ngx_chain_t *in)
    {
        if(!in)                                     //空数据链无须处理
        {
            return NGX_OK;
        }

        if( this_module::ctx().empty(r) ||           //检查环境数据
            !this_module::data(r).do_hook)           //没有标志位则不处理
        {
            return this_filter::next(r, in);
        }

        auto& pr_ctx_data = this_module::data(r->parent); //获取父请求的 ctx

        auto rc = ngx_chain_add_copy(                //拷贝到父请求模块的 ctx 里
            r->pool, &pr_ctx_data.body, in);
        NgxException::require(rc);

        return NGX_OK;                               //结束过滤链表
    }

```

函数里我们调用了 Nginx 的 `ngx_chain_add_copy()` 函数，它并不真正拷贝数据，只是在目标链表里新建链表节点，引用原链表里的数据块，没有效率损失。

在回传之后我们必须直接返回 `NGX_OK`，终止过滤链的执行，否则数据会走到 `ngx_http_postpone_filter_module`，从主请求里输出，破坏我们的回传计划。

10.4.4 模块集成类

`ndg_data_hook_module` 模块没有配置数据，所以集成类非常简单，只需设置 `post-configuration` 函数指针初始化过滤链表：

```

class NdgDataHookInit final
{
public:
    typedef NdgDataHookHandler    handler_type;    //简化类型定义
    typedef NdgDataHookInit       this_type;

public:
    static ngx_http_module_t* ctx()                //函数指针表
    {
        static ngx_http_module_t c =
        {
            NGX_MODULE_NULL(1),
            &handler_type::init,                      //初始化过滤链表
            NGX_MODULE_NULL(6),                      //没有配置数据的创建函数
        };
    }

```

```

    return &c;
}
public:
    static const ngx_module_t& module()
    {...}
};
//省略模块初始化代码

```

10.4.5 编译脚本

ndg_data_hook_module 模块在编译进 Nginx 时需要小心，因为它会中途截断过滤链表的执行，所以链表的顺序很重要，在它之后的 filter 模块都会失效。

模块的编译脚本 config 的内容是：

```

ngx_addon_name=ndg_data_hook_module
ngx_module_type=HTTP_FILTER
ngx_module_name=ndg_data_hook_module
ngx_module_srcs="$ngx_addon_dir/ModNdgDataHook.cpp"

. auto/module

```

configure 脚本会把 filter 模块安排在 ngx_http_copy_filter_module 和 ngx_http_headers_filter_module 之间（参见 7.3.3 节），位置正好在 ngx_http_postpone_filter_module 之前，所以它可以正常工作。

但如果有多多个第三方 filter 模块，那么就需要协调 ndg_data_hook_module 与它们之间的关系。例如，如果想让数据被 9.10 节开发的 ndg_footer_module 处理后再回传，那么就要执行下面的命令：

```

./configure \
    --add-module=path/to/datahook \
    --add-module=path/to/footer \
    #配置 Nginx 的编译选项
    #hook 模块在前
    #footer 模块在后

```

这样生成的 ngx_modules 数组就是：

```

ngx_module_t *ngx_modules[] = {
    ...
    &ngx_http_postpone_filter_module, //Nginx 数据整理模块
    &ngx_http_ssi_filter_module,
    &ngx_http_charset_filter_module,
    &ngx_http_userid_filter_module,
    &ngx_http_headers_filter_module,

    &ndg_data_hook_module, //过滤链表的后位置

```

```

&ndg_footer_module,                                //过滤链表的前位置

ngx_http_copy_filter_module,
ngx_http_range_body_filter_module,
ngx_http_not_modified_filter_module,                //过滤链表头节点
NULL
};

```

10.5 在模块里使用子请求

本节将使用前两节实现的 C++ 工具类和数据回传模块，示范 Nginx 子请求的用法。

10.5.1 模块设计

示范子请求用法的模块设计如下：

- 模块名是 `ndg_subrequest_module`，是内容处理模块；
- 模块使用 `content handler` 的方式注册处理函数；
- 配置指令 `ndg_subrequest_loc`，确定发起子请求使用的 `uri`，即 `location`；
- 配置指令 `ndg_subrequest_args`，确定发起子请求使用的 `args`；
- 父请求把子请求的响应头和响应体均原样输出。

10.5.2 配置信息类

`NdgSubrequestConf` 定义模块所需的配置信息，它有两个 `ngx_str_t` 成员，存储发起子请求需要的 `uri` 和 `args`：

```

class NdgSubrequestConf final
{
public:
    typedef NdgSubrequestConf this_type;                //简化类型定义
public:
    NdgSubrequestConf() = default;
    ~NdgSubrequestConf() = default;
public:
    ngx_str_t loc;                                       //子请求 uri
    ngx_str_t args;                                      //子请求 args
public:
    static void* create(ngx_conf_t* cf)                 //创建结构体对象
    {
        return NgxPool(cf).alloc<this_type>();
    }
}

```

```

... //其他成员函数，见后
};

NdgSubrequestConf 还实现了 merge() 函数，为字符串提供默认值：

public:
    static char* merge(ngx_conf_t *cf, void *parent, void *child)
    {
        auto prev = reinterpret_cast<this_type*>(parent);
        auto conf = reinterpret_cast<this_type*>(child);

        NgxValue::merge(conf->loc, prev->loc, ngx_string("/echo"));
        NgxValue::merge(conf->args, prev->args, ngx_null_string);

        return NGX_CONF_OK;
    }
};

```

10.5.3 业务逻辑类

ndg_subrequest_module 需要发起子请求，获取子请求产生的数据，然后再生成自己的数据，主要功能都在回调函数类 NdgSubrequestCallback 里实现。

子请求回调函数

子请求回调函数要实现的功能较为简单，只是把子请求的错误码回传给父请求，放置在 ndg_data_hook_module 模块的 ctx 里，不使用 data 参数：

```

class NdgSubrequestCallback final
{
public:
    typedef NdgDataHookModule hook_module; //简化类型定义
public:
    //子请求结束时的回调函数，注意参数 r 表示的是子请求对象，data 参数不使用
    static ngx_int_t sub_post(ngx_http_request_t* r, void* data, ngx_int_t rc)
    {
        auto& pr_ctx_data = hook_module::data(r->parent); //获取父请求的模块 ctx
        pr_ctx_data.state(rc); //回传错误码

        return NGX_OK; //回调函数执行成功
    }
    ... //父请求回调函数见后
};

```

父请求回调函数

在父请求回调函数里需要从 `ndg_data_hook_module` 模块获取回传的状态码、响应头和响应体，然后和其他 `content handler` 模块一样，构造响应数据，调用 `send()` 发给客户端：

```
public:
//父请求结束时的回调函数，注意参数 r 表示的是父请求对象
static ngx_int_t parent_post(ngx_http_request_t* r)
{
    auto& ctx_data = hook_module::data(r);           //获取模块的 ctx

    if(ctx_data.status != NGX_HTTP_OK)               //检查子请求的处理状态
    { return NGX_HTTP_INTERNAL_SERVER_ERROR; }

    ngx_str_t msg = ngx_string("subrequest");         //要输出的一个字符串

    NgxChain chain = ctx_data.body;                  //子请求的响应数据
    auto len = msg.len + chain.size();                //计算输出总长度

    NgxResponse resp(r);                             //准备发送响应数据

    resp.length(len);                                //设置内容长度

    NgxHeadersOut h(ctx_data.headers);               //子请求的头信息

    h.list().visit(                                   //遍历子请求的头信息
        [&](ngx_table_elt_t& x)                      //lambda 表达式
        {
            resp.headers().add(x);                    //添加到输出响应头
        });

    for(auto& ch : chain)                             //遍历子请求数据
    {
        if(ch.data().special())                       //不发送控制信息
        { continue; }

        if(ch.data().last())                          //去掉 last 信息
        { ch.data().finish(false); }

        resp.send(ch.data());                         //发送子请求数据
    }

    resp.send(&msg);                                  //发送本请求的信息
    resp.eof();                                       //响应数据结束
}
```

```

        return NGX_OK; //处理成功,返回 NGX_OK
    }

```

代码里要注意的是对子请求数据的处理,首先要检查子请求的处理状态。如果不是 `NGX_HTTP_OK` 就意味着子请求处理发生了错误,之后可以从 `ctx` 的 `headers` 和 `body` 成员分别得到子请求返回的头信息和数据。

由于 `ndg_data_hook_module` 模块把子请求的数据原封不动地“钩”到了父请求里,所以数据链里可能会有 `sync`、`flush`、`eof` 等用于控制的特殊缓冲区,必须要对它们做特殊处理,否则可能会影响父请求的正常输出。

回调函数最后需要返回 `NGX_OK` 或 `NGX_HTTP_INTERNAL_SERVER_ERROR` 等错误码,它会传递给 `ngx_http_finalize_request()`,最终结束整个请求。

发起子请求

完成回调函数类后,我们用 `NgxSubRequest` 把它组装起来,在 `handler` 里发起子请求:

```

class NdgSubrequestHandler final
{
public:
    typedef NdgSubrequestHandler  this_type; //简化类型定义
    typedef NdgSubrequestCallback callback_type;
    typedef NdgDataHookModule     hook_module;
    typedef NdgSubrequestModule   this_module;

    typedef NgxSubRequest<callback_type> this_subrequest;
public:
    static ngx_int_t handler(ngx_http_request_t *r)
    {
        NgxRequest req(r);

        if(!req.method(NGX_HTTP_GET)) //检查请求方法
        { return NGX_HTTP_NOT_ALLOWED; }

        auto& cf = this_module::conf().loc(r); //获取配置

        this_subrequest(r).create(cf.loc, cf.args); //使用配置创建子请求

        hook_module::data(r).hooking(); //要求执行数据回传

        return NGX_DONE; //返回 NGX_DONE
    }
}

```



```
};
```

`NdgSubrequestHandler` 首先要做的就是用回调函数类实例化 `NgxSubRequest`，组装好调用子请求必须的两个回调函数，然后使用 `create()` 创建子请求（这时子请求还没有开始执行）。要想把子请求的数据回传到父请求，就需要调用 `ndg_data_hook_module` 模块的 `hooking()` 函数，设置回传标志位。

函数结束时必须返回 `NGX_DONE`，暂停请求的处理，等待子请求结束时被“激活”，然后在回调函数 `parent_post()` 里执行后续的操作。

10.5.4 测试验证

`ndg_subrequest_module` 的模块集成和编译与之前开发的模块基本相同，这里不再列出详细的实现代码，直接进入最后的测试验证环节。

在 Nginx 的配置文件里定义子请求访问的 location:

```
location /hello {                                #子请求访问的 location
    ndg_echo "hello";                            #输出字符串“hello”

    ndg_header x-name chrono;                    #加入自定义头信息
    ndg_footer "\n";                             #响应数据末尾添加换行
}
```

然后再定义发起子请求的 location:

```
location /sub {                                  #发起子请求的 location
    ndg_subrequest_loc "/hello";                 #访问"/hello"
    ndg_subrequest_args "chrono";                #添加子请求参数

    ndg_footer "\n";                             #响应数据末尾添加换行
}
```

使用 curl 访问 /sub，模块会使用子请求获取 /hello 的数据，整合后输出:

```
curl http://localhost/sub -v                    #执行 curl 命令

> GET /sub HTTP/1.1                             #请求头，访问/sub
>
< HTTP/1.1 200 OK                               #请求处理成功
< Server: nginx/1.12.0
< Content-Length: 24
< x-name: chrono                                #子请求里的头信息
<
chrono,hello                                     #子请求的响应数据
```

subrequest

#父请求的响应数据

10.6 总结

本章介绍了 Nginx 里的子请求机制，能够把服务器里的 location 变成一个个的函数调用，使用子请求可以组合这些 location，完成复杂的业务逻辑，让 Nginx 超越简单的 Web 服务器成为应用服务器。

沿用之前的 Nginx 宫殿比喻，子请求就像是宫殿里穿梭不息的仆人，连接起了各个房间（模块），他们在这些房间里搜集原料，最后生产出精美的工艺品（响应数据）。

子请求的设计思想是“分而治之”，很像是解决数学难题，把主问题分解为若干个易于解决的小问题，每个小问题都解决后整个问题也就解决了。任何复杂的业务都是如此，总是可以分解出一些粒度略小的任务，这些任务可以用一个子请求来表示。分解操作可以递归进行，直至每个任务都能够用一个简单的 HTTP 请求完成。最后我们整理子请求的处理结果，进行分类汇总等加工操作，就完成了整个业务。

子请求本质上与普通请求没有区别，只是它由 Nginx 在内部发起，省略了 TCP 连接和请求数据解析的步骤，而且可以并发多个。Nginx 创建子请求默认是 GET 方法，大部分参数也都拷贝自父请求，但用户可以任意修改子请求的 method/uri/headers/body 等参数，定制程度很高。

与创建子请求和调度子请求的设计相比，Nginx 对子请求数据的处理显得有些“草率”。ngx_http_postpone_filter_module 模块虽然能够把子请求的数据传回父请求，但没有提供回调函数，我们无法对数据做额外的处理，所以实用意义不是很大。

但 ngx_http_postpone_filter_module 使用子请求数据的思路是值得借鉴的，所以本章开发了数据回传模块 ndg_data_hook_module，能够把子请求的响应数据全部“钩”到父请求里，这样父请求就可以像函数调用一样随意地使用配置文件里的 location。

本章的最后我们使用 C++ 实现了一个调用子请求的示范模块，读者可以参考它开发出应用在实际业务里的子请求调用。

第 11 章

Nginx 变量

本书第 1 章曾简单地介绍了 Nginx 的变量，它与配置文件里固定的参数不同，值是可变的。用户可以在 Nginx 配置文件里以“\$var”的形式获取 Nginx 预定义的各种信息，例如 \$uri、\$args、\$remote_addr，也可以使用 set 和 map 等指令来定义新的变量，善用变量能够极大地增加 Nginx 配置的灵活性，让配置文件更像是一种小型语言。

Nginx 变量表面上与编程语言的变量很相似，但实际上它只是 Nginx 内部存储的一些字符串，是模块对外导出的一段数据。变量不仅可以用于配置文件，还可以用在 C/C++ 代码里，用作模块间的简单通信，或者部分替代配置指令来设置模块的功能参数。

Nginx 的变量可以用在 HTTP/TCP 处理^①，它们通常都与具体的请求相关——这是显而易见的，因为如 URI、IP 地址、响应时间等每个请求都不相同。

本章主要介绍 HTTP 框架里的变量，Stream 框架里变量的原理和用法与它基本相同。

11.1 结构定义

Nginx 定义了两个结构体来实现变量机制：变量值 ngx_variable_value_t 和变量访问对象 ngx_http_variable_t。

11.1.1 变量值

ngx_variable_value_t 与 ngx_str_t 类似，表示一个内存里的字符串空间，但它增加了一些属性成员：

^① 自 Nginx 1.11.2 开始 stream 框架才支持使用变量。

```
//定义在 core/nginx_string.h
typedef struct {
    unsigned    len:28;                //变量字符串长度

    unsigned    valid:1;               //变量是否有效
    unsigned    no_cacheable:1;       //变量值是否允许缓存
    unsigned    not_found:1;          //变量是否存在
    unsigned    escape:1;              //变量是否被 escape 处理

    u_char*     data;                 //变量的内存地址
} ngx_variable_value_t;
```

ngx_variable_value_t 只是一个简单的字符串值，并不与变量发生直接关系。它的结构比较简单，可以认为是 ngx_str_t 的一个变体，成员 len 只使用了 28 位来表示长度，余下的 4 位作为标志位，表示变量的几种状态。

HTTP 框架里又对 ngx_variable_value_t 做了重定义：

```
//定义在 http/nginx_http_variables.h
typedef ngx_variable_value_t  ngx_http_variable_value_t;
```

11.1.2 变量访问对象

Nginx 在 ngx_variable_value_t 之上定义了 ngx_http_variable_t 结构体，为变量值的读写增加了一个间接层，它表示真正的 Nginx 变量，使用 get/set 函数而不是简单的字符串来访问变量值，增加了处理的灵活性。

ngx_http_variable_t 定义如下：

```
//定义在 http/nginx_http_variables.h
typedef struct ngx_http_variable_s  ngx_http_variable_t;

struct ngx_http_variable_s {
    ngx_str_t          name;           //变量名
    ngx_http_set_variable_pt  set_handler; //设置变量值函数
    ngx_http_get_variable_pt  get_handler; //获取变量值函数
    uintptr_t          data;           //set/get 函数使用的辅助参数
    ngx_uint_t         flags;          //变量属性标志位
    ngx_uint_t         index;          //变量所在的数组序号
};
```

ngx_http_variable_t 结构体里的关键成员是 get_handler 和 set_handler，这两个函数会根据 ngx_http_request_t 和 data 来计算变量的值。如果 Nginx 内部已经有值（例如 \$uri、\$args），那么就填充 ngx_http_variable_value_t 的 len/data 直接

输出，否则就需要在内存池里分配内存，拼接好字符串再输出。所以，通过 Nginx 变量来获取信息是有一定计算成本的。

get/set 函数的声明是：

```
typedef void (*ngx_http_set_variable_pt) (ngx_http_request_t *r,
                                          ngx_http_variable_value_t *v, uintptr_t data);
typedef ngx_int_t (*ngx_http_get_variable_pt) (ngx_http_request_t *r,
                                              ngx_http_variable_value_t *v, uintptr_t data);
```

结构体里的 flags 成员可以取值为下面的宏，设置变量的属性：

```
#define NGX_HTTP_VAR_CHANGEABLE    1           //是否可以修改
#define NGX_HTTP_VAR_NOCACHEABLE  2           //是否允许缓存
#define NGX_HTTP_VAR_INDEXED       4           //是否已在请求对象里被索引
#define NGX_HTTP_VAR_NOHASH        8           //是否允许 hash
#define NGX_HTTP_VAR_WEAK          16          //“弱变量”
#define NGX_HTTP_VAR_PREFIX        32          //是否有"http_"等前缀
```

11.1.3 变量的存储

Nginx 把变量对象统一存储在 ngx_http_core_module 配置数据结构的 variables 成员里：

```
typedef struct {
    ...
    ngx_array_t          variables;           //存储所有的变量对象
    ngx_array_t          prefix_variables;    //存储有前缀的变量对象
    ngx_hash_t           variables_hash;      //用于快速查找的 hash 表
    ngx_hash_keys_arrays_t* variables_keys;   //临时存储 hash key
    ...
} ngx_http_core_main_conf_t;
```

variables 是一个动态数组，它的元素是 ngx_http_variable_t，在配置解析结束时 Nginx 会调用 ngx_http_variables_init_vars() 函数，把所有模块定义的变量集中存储在这里。所以，变量的定义是跨 server 和 location 生效的。

variables_hash 是一个散列表，同样在 ngx_http_variables_init_vars() 函数里被初始化，可以使用函数 ngx_hash_find() 快速查找定位变量的位置。

11.1.4 请求结构体

请求结构体 ngx_http_request_t 为变量值提供了独立的存放空间，因此每个请求获取的变量值都是独立的（即变量是请求相关的）：

```

struct ngx_http_request_s {
    ...
    ngx_http_variable_value_t*    variables;    //存放所有可能的变量值
    ...
};

```

在函数 `ngx_http_create_request()` 创建请求对象时，会依据 `cmcf->variables` 数组为变量值分配内存：

```

ngx_http_request_t *
ngx_http_create_request(ngx_connection_t *c)    //创建请求
{
    ...
    r->variables = ngx_palloc(r->pool,          //数组长度与 cmcf 里的相同
                              cmcf->variables.nelts*sizeof(ngx_http_variable_value_t));
    ...
};

```

可以看到，请求里的变量值数组 `variables` 与配置里的变量对象数组 `variables` 是完全对应的。

但为了节约内存，`r->variables` 并不会主动存放变量值，只有当用户在实际访问变量时才会调用变量的 `get_handler` 函数，取出变量值并放入数组。

需要注意的是，对于子请求来说，变量值数组是与父请求共享的，而不是独立的，在创建子请求的函数 `ngx_http_subrequest()` 里代码是：

```

sr->variables = r->variables;    //直接引用父请求的变量数组指针

```

这既是优点也是缺点，优点是节约内存，不会因为创建大量子请求而增加内存消耗，缺点则是某个子请求可能修改了变量而父请求并不知情，导致预想之外的错误。

如果想要父请求和子请求的变量之间互不影响，就需要在子请求创建后修改它自己的 `variables` 成员，创建一个新数组，然后拷贝父请求的数组内容，制作变量的副本。

11.2 运行机制

Nginx 的每个 http 模块都可以在配置解析时向框架注册变量，导出模块专有的信息，变量统一存储在 `ngx_http_core_module` 的 `cmcf->variables` 动态数组里。

在 HTTP 请求处理阶段，我们可以从中获取变量对象，使用它的 `get_handler` 和 `set_handler` 函数操作变量值。

11.2.1 注册变量

函数 `ngx_http_add_variable()` 是 Nginx 变量机制的核心函数，它创建一个命名的变量访问对象 `ngx_http_variable_t`。

`ngx_http_add_variable()` 的声明是：

```
ngx_http_variable_t *ngx_http_add_variable(ngx_conf_t *cf,
                                             ngx_str_t *name, ngx_uint_t flags);
```

http 模块通常使用一个静态数组存放变量定义，在配置解析前的 `preconfiguration` 时调用 `ngx_http_add_variable()`，把它们添加进 Nginx 内部。例如 `ngx_http_upstream_module` 里的代码是：

```
//代码在 http/ngx_http_upstream.c 里
static ngx_http_variable_t ngx_http_upstream_vars[] = {
    { ngx_string("upstream_addr"), NULL, //变量$upstream_addr
      ngx_http_upstream_addr_variable, 0, //变量的 get 函数
      NGX_HTTP_VAR_NOCACHEABLE, 0 }, //访问标志位
    ... //其他变量定义
    { ngx_null_string, NULL, NULL, 0, 0, 0 } //空变量对象表示数组结束
};

for (v = ngx_http_upstream_vars; v->name.len; v++) { //遍历数组
    var = ngx_http_add_variable(cf, &v->name, v->flags); //添加变量
    var->get_handler = v->get_handler; //设置 get 函数
    var->data = v->data;
}
```

http 块解析时 Nginx 会调用所有模块的 `preconfiguration` 函数注册变量，然后执行 `ngx_http_variables_init_vars()` 函数，把变量存入 `cmcf->variables`。

11.2.2 获取变量

函数 `ngx_http_get_variable()` 可以访问 Nginx 变量值，它使用变量名和 hash key 在 `ngx_http_core_module` 里查找已经添加的变量，再调用 `get_handler` 获取变量值。^①

`ngx_http_get_variable()` 的声明是：

^① `ngx_http_get_variable()` 函数里对 “http_”、“sent_http_” 等变量前缀有特殊处理。

```
ngx_http_variable_value_t* ngx_http_get_variable(ngx_http_request_t *r,
                                                  ngx_str_t *name, ngx_uint_t key);
```

函数返回变量值对象，但因为变量可能不存在（未定义），所以需要检查指针是否有效和 `not_found` 成员。

11.2.3 修改变量

修改变量前必须要在 `cmcf->variables_hash` 里找到变量，如果变量具有“不可修改”的属性 (`NGX_HTTP_VAR_CHANGEABLE`)，那么就不能变动变量值。

查找变量需要使用 `ngx_hash_find()` 函数，它的声明是：

```
void *ngx_hash_find(ngx_hash_t *hash,
                   ngx_uint_t key, u_char *name, size_t len);
```

注意它的返回值类型，是 `void*`，必须转型为 `ngx_http_variable_t*`。

如果变量有 `set_handler` 成员，那么我们就调用 `set_handler` 修改变量值。

如果变量没有 `set_handler` 成员，但属性是 `NGX_HTTP_VAR_INDEXED`，这意味着它位于请求的变量值数组 `r->variables` 里，就可以直接修改 `ngx_http_variable_value_t` 里的 `len` 和 `data` 成员。

11.3 复杂变量

基于变量 Nginx 还实现了更高级的“脚本”（script）功能，可以让模块支持复杂的变量组合，解析处理内部含有一个或多个变量定义的字符串——也就是“脚本”。

11.3.1 结构定义

Nginx 把含有多个待解析的变量的字符串称为“复杂变量”，结构体 `ngx_http_complex_value_t` 定义是：

```
typedef struct {
    ngx_str_t      value;           //复杂变量的值，即字符串
    ngx_uint_t*    flushes;         //内部成员，暂无须关心
    void*          lengths;
    void*          values;
} ngx_http_complex_value_t;
```

复杂变量（脚本）都是在配置文件里定义的，必须在配置解析阶段执行“编译”，之后才

能正确得到变量值，这需要另外一个结构体 `ngx_http_compile_complex_value_t`：

```
typedef struct {
    ngx_conf_t*      cf;                //Nginx 的配置结构体指针
    ngx_str_t*        value;             //配置文件里的原始字符串
    ngx_http_complex_value_t* complex_value; //编译后的输出结果，即复杂变量

    unsigned          zero:1;            //内部成员，暂无须关心
    unsigned           conf_prefix:1;
    unsigned           root_prefix:1;
} ngx_http_compile_complex_value_t;
```

11.3.2 运行机制

复杂变量与普通字符串的用法类似，也需要在模块的配置结构体里提供一个存储位置，但它不能使用简单的 `ngx_str_t`，而必须使用 `ngx_http_complex_value_t`，并且在指令解析时调用函数 `ngx_http_complex_value()` 进行“编译”。

函数 `ngx_http_complex_value()` 的声明是：

```
ngx_int_t ngx_http_compile_complex_value( //指令解析时“编译”复杂变量
    ngx_http_compile_complex_value_t *ccv);
```

它只有一个参数 `ccv`，需要我们自己填充 `ccv` 里的 `cf`、`value` 字段，而 `ccv->complex_value` 字段就是模块配置结构体里的复杂变量对象。

在运行时，调用函数 `ngx_http_compile_complex_value()` 就可以获得复杂变量的实际内容：

```
ngx_int_t ngx_http_complex_value( //运行时获取变量值
    ngx_http_request_t *r,
    ngx_http_complex_value_t *val, ngx_str_t *value);
```

参数 `val` 是配置结构体里的复杂变量对象（必须在之前已经“编译”过），变量的内容则在 `value` 里输出。

11.4 C++封装

Nginx 变量的内部实现比较复杂，但接口却比较简单，因此 C++封装类的代码也就很容易理解。

11.4.1 ngxVariableValue

类 ngxVariableValue 代理了 ngx_http_variable_value_t，提供对变量值的基本操作，定义如下：

```
class NgxVariableValue final : public NgxWrapper<ngx_variable_value_t>
{
public:
    typedef NgxWrapper<ngx_variable_value_t> super_type;
    typedef NgxVariableValue this_type;
public:
    NgxVariableValue(ngx_variable_value_t* v): super_type(v) {}
    ~NgxVariableValue() = default;
public:
    ... //成员函数见后
};
```

读取变量值

调用 ngx_http_get_variable() 得到的变量值不一定是有效的，所以需要进行判断：

```
public:
    bool valid() const //变量值是否有效
    {
        return get() && get()->valid && !get()->not_found; //检查标志位
    }
```

使用成员 len 和 data 就可以获得变量值，它们可以转换为通用的 ngx_str_t：

```
public:
    ngx_str_t str() const //获取变量值字符串
    {
        return valid()?
            ngx_str_t{get()->len, get()->data}: //是否有效
            ngx_str_t ngx_null_string; //返回字符串
    } //无效返回空字符串
```

修改变量值

操作 ngx_http_variable_value_t 里的成员就可以修改变量值，但同时还要设置 valid、not_found 等标志位，否则即使修改字符串也是无效的：

```
public:
    void set(ngx_str_t* str, bool clear = false) //修改变量值
    {
        get()->len = clear ? 0 : str->len; //设置字符串长度
```

```

get()->data = clear ? nullptr : str->data;           //设置字符串内容

get()->valid = !clear;                             //变量有效标志位
get()->not_found = clear;                          //可以找到标志位
get()->no_cacheable = false;                       //可以缓存
}

```

在 `set()` 函数里我们使用了一个默认参数 `clear`，它如果置为 `true` 就会清空变量值。

11.4.2 ngxVariable

类 `ngxVariable` 封装了读写变量值的操作，定义如下：^①

```

class ngxVariable final
{
public:
    ngxVariable(ngx_http_request_t* r, string_ref_type name):
        m_r(r), m_pool(r)                                //内存池初始化
    {
        m_name = m_pool.dup(name);                      //拷贝一份字符串并计算 hash key
        m_key = ngx_hash_strlow(m_name.data, m_name.data, m_name.len);
    }

    ~ngxVariable() = default;

private:
    ngx_http_request_t*    m_r = nullptr;               //请求对象
    ngx_pool_t             m_pool;                      //内存池
    ngx_str_t              m_name = ngx_null_string;    //小写的变量名
    ngx_uint_t             m_key = 0;                  //hash key
public:
    ...                                                 //其他成员函数见后
};

```

`ngxVariable` 的成员 `m_name` 和 `m_key` 保存了 Nginx 在查找变量时必需的小写化名字和 hash key，用于在 `get` 和 `set` 时快速定位变量。

读取变量值

成员函数 `get()` 调用 `ngx_http_get_variable()` 直接得到变量值对象，返回变量的字符串内容：

```

public:

```

^① `ngxVariable` 还重载了 `operator<<` 提供流输出操作，此处未列出，读者可参考 GitHub 资源。

```

ngx_str_t get() const //读变量值
{
    NgxVariableValue vv = ngx_http_get_variable(m_r, //获取变量值对象
        const_cast<ngx_str_t*>(&m_name), m_key); //注意常量转换

    return vv.str(); //返回 Nginx 字符串
}

```

因为 NgxVariableValue 已经对变量值的有效性做了检查,所以如果变量不存在函数就会返回一个空字符串。

设置变量值

成员函数 set() 首先在配置结构体里检查变量是否存在,然后根据变量是否有 set_handler 决定具体的修改方式:

```

public:
    bool set(string_ref_type value) const //写变量值
    {
        auto& cmcf = NgxHttpCoreModule::instance().conf().main(m_r);

        //在配置结构体里使用散列表查找变量是否存在
        auto p = ngx_hash_find(&cmcf.variables_hash,
            m_key, m_name.data, m_name.len);

        if(!p) //不存在则写失败
        { return false; }

        auto v = (ngx_variable_t*)(p); //转型为正确的指针类型

        if(!(v->flags & NGX_HTTP_VAR_CHANGEABLE)) //不允许修改则写失败
        { return false; }

        auto val = m_pool.dup(value); //内存池拷贝一份字符串

        NgxVariableValue vv = v->set_handler ? //是否有 set_handler
            m_pool.alloc<ngx_http_variable_value_t>() : //有则新建一个变量值对象
            ((v->flags & NGX_HTTP_VAR_INDEXED) ? //是否已经在请求里被索引
                &m_r->variables[v->index] : nullptr); //已索引则直接使用

        if(!vv) //变量值创建失败或不存在
        { return false; } //写失败

        vv.set(val, value.empty()); //设置变量值, empty 则清空

        if(v->set_handler) //有 set_handler 则调用

```

```

{   v->set_handler(m_r, vv, v->data); }           //用变量值设置

return true;                                     //写成功
}

```

便捷操作

NgxVariable 还重载了两个操作符，转型操作符相当于 `get()`，`operator=` 相当于 `set()`：

```

public:
    operator ngx_str_t() const                    //转化为 ngx_str_t 类型
    {
        return get();
    }

    void operator=(string_ref_type value) const    //重载赋值操作符
    {
        set(value);
    }

```

11.4.3 NgxVarManager

访问 Nginx 变量必须要结合具体的 HTTP 请求，类 NgxVarManager 保存了 `ngx_http_request_t` 指针，用类似字典的 `vars["name"]` 方式提供便捷的读写操作。

NgxVarManager 内部保存了一个请求对象 `m_r`，并重载了 `operator[]`，可以方便地产生 NgxVariable 对象：

```

class NgxVarManager final
{
public:
    NgxVarManager(ngx_http_request_t* r): m_r(r) {}           //构造获得请求对象
    ~NgxVarManager() = default;

public:
    NgxVariable operator[](string_ref_type name) const
    {
        return NgxVariable(m_r, name);                       //返回代理对象
    }

private:
    ngx_http_request_t* m_r = nullptr;                       //请求结构体指针
};

```

11.4.4 NgxVariables

类 NgxVariables 负责模块变量的初始化，它模仿了 Nginx 的惯用手法，使用模板参数里提供的变量数组简化了模块的变量添加工作：

```
template<ngx_http_variable_t*(*get_vars)()>           //模板参数是函数指针
class NgxVariables final                               //用于添加变量
{
public:
    static ngx_int_t init(ngx_conf_t *cf)
    {
        if(!get_vars)                                //不能是空指针
        { return NGX_OK; }

        for (auto v = get_vars(); v->name.len; ++v)    //遍历数组
        {
            var_type var = ngx_http_add_variable(cf, &v->name, v->flags);

            NgxException::fail(!var);                  //不能是空指针

            var->get_handler = v->get_handler;          //添加 get 函数
            var->data = reinterpret_cast<uintptr_t>(v->data);
        }

        return NGX_OK;
    }
};
```

NgxVariables 的模板参数 get_vars 是一个函数指针，我们可以把变量数组的定义和 get 函数都封装在一个类里，再用静态成员函数实例化 NgxVariables，最后在 pre-configuration 时调用 init() 即可。

11.4.5 NgxComplexValue

类 NgxComplexValue 封装了复杂变量的初始化和获取操作，定义如下：

```
class NgxComplexValue final
{
private:
    ngx_str_t          m_source;           //配置文件里的原始字符串
    ngx_http_complex_value_t m_cv;        //复杂变量对象
public:
    ...                                    //其他成员函数见后
};
```

NgxComplexValue 里的核心成员是 `m_cv`，它保存了复杂变量对象。

成员函数 `init()` 在指令解析时“编译”脚本，初始化复杂变量对象：

```
public:
    void init(ngx_conf_t* cf, ngx_str_t* source)           //初始化复杂变量
    {
        m_source = *source;                               //保存一份原始字符串

        ngx_http_compile_complex_value_t ccv;             //“编译”用的辅助对象
        NgxValue::memzero(ccv);                           //需要先清空

        ccv.cf = cf;                                       //设置配置对象
        ccv.value = source;                                //设置原始字符串
        ccv.complex_value = &m_cv;                         //设置复杂变量对象

        auto rc = ngx_http_compile_complex_value(&ccv); //设置完毕，执行“编译”
        NgxException::require(rc);                         //如果出错则抛出异常
    }
```

成员函数 `str()` 在运行时获取复杂变量的值：

```
public:
    ngx_str_t str(ngx_http_request_t* r)                 //获取复杂变量的值
    {
        ngx_str_t value = ngx_null_string;               //用于输出的字符串

        auto rc = ngx_http_complex_value(r, &m_cv, &value); //获取复杂变量的值
        NgxException::require(rc);                       //如果出错则抛出异常

        return value;                                     //返回字符串
    }
```

11.5 在模块里使用变量

`NgxVariables` 和 `NgxVarManager` 这两个类封装了 Nginx 变量相关的各种操作，利用它们就能够在模块里处理 Nginx 变量。

11.5.1 添加变量

作为示例，我们实现两个新变量：

- 当前请求的方法名，即 `r->method_name`，导出为 `$current_method`；

■ 当天日期，使用 3.6.6 节的 `NgxDatetime`，导出为 `$today` 变量。

类 `NdgVariablesHandler` 封装了变量数组和 `get` 函数：

```
class NdgVariablesHandler final
{
public:
    typedef NdgVariablesHandler this_type;
public:
    static ngx_http_variable_t* get_vars() //变量定义数组
    {
        static ngx_http_variable_t vars[] = { //数组里定义了一个变量

            { ngx_string("today"), nullptr, //变量名字是 today
              &this_type::get_today, 0, //get 函数
              0, 0 }, //可以缓存，不能修改

            { ngx_string("current_method"), nullptr, //变量名字
              &this_type::get_current_method, 0, //get 函数
              NGX_HTTP_VAR_NOCACHEABLE, 0 }, //不能缓存，不能修改

            { ngx_null_string, nullptr, nullptr, 0, 0, 0 } //空对象表示数组结束
        };

        return vars; //返回变量数组
    }
public:
    static ngx_int_t get_current_method( //get 函数
        ngx_http_request_t *r, ngx_http_variable_value_t *v, uintptr_t data)
    {
        NgxVariableValue(v).set(r->method_name); //直接用请求的成员设置变量值

        return NGX_OK;
    }

    static ngx_int_t get_today( //get 函数
        ngx_http_request_t *r, ngx_http_variable_value_t *v, uintptr_t data)
    {
        auto str = NgxDatetime::today(); //获取当天日期

        NgxVariableValue(v).set(str); //设置变量值

        return NGX_OK;
    }
};
```


在模块的 `ngx_http_module_t` 里设置函数指针 `preconfiguration` 为 `NgxVariables::init()`，即可完成变量的添加工作：

```
//实例化变量初始化类
typedef NgxVariables<&handler_type::get_vars> these_variables;

static ngx_http_module_t* ctx()
{
    static ngx_http_module_t c = //配置阶段的函数指针表
    {
        &these_variables::init, //preconfiguration
        NGX_MODULE_NULL(7) //其余七个是空指针
    };

    return &c;
}
```

11.5.2 读写变量

`NgxVarManager` 集中管理了 `Nginx` 的所有变量，只要用请求对象构造后就可以使用 `operator[]` 访问任意变量，很像 `std::map`，示范代码如下：

```
NgxVarManager var(r); //构造 NgxVarManager 对象

cout << var["current_method"]<< endl; //访问$current_method
cout << var["var1"]<< endl; //访问变量$var1

var["var1"] = "1234567"; //修改变量$var1 的值
cout << var["var1"]<< endl; //输出变量$var1 的值
```

`NgxVarManager` 能够非常方便地读写变量，所以模块可以使用指令 `set` 来定义变量，配置所需的各种参数，部分地取代配置指令的作用，不需要再额外编写解析指令的代码。

不过变量的读写有少量的计算和内存使用成本，需要对此进行评估。如果能够直接从请求结构体里获取就最好直接访问（例如 `$current_method` 可以用 `r->method`），尽量不要频繁使用变量导致影响性能。

11.6 在模块里使用复杂变量

我们修改 8.10 节的 `echo` 模块，为它添加复杂变量（脚本）的支持。

11.6.1 配置信息类

之前的 `NdgEchoConf` 里使用字符串成员来保存配置信息，只能支持简单字符串，现在我们可以把它改成 `NgxComplexValue`，也就是：

```
class NdgEchoConf final                                //模块的配置结构体
{
public:
    NgxComplexValue var;                                //存储配置文件里的脚本
    ...                                                  //其他成员
};
```

11.6.2 业务逻辑类

由于 `NgxComplexValue` 封装了 Nginx 的底层接口，所以复杂变量的使用非常简单，与一个普通字符串没有太多差异：

```
auto& cf = this_module::conf().loc(r);                //获取模块的配置信息
auto str = cf.var.str(r);                              //获取变量内容
...                                                    //设置请求头，发送数据……
```

11.6.3 模块集成类

在指令解析函数里我们不能使用 `ngx_conf_set_str_slo()` 解析指令，因为它只能处理简单字符串，而必须使用 C++ 封装类调用 `ngx_http_compile_complex_value()` 来初始化复杂变量：

```
static char* set_echo(ngx_conf_t* cf, ngx_command_t* cmd, void* conf)
{
    NgxStrArray args(cf->args);                        //获取指令字符串数组
    auto& lcf = conf_type::cast(conf);                 //配置信息结构体类型转换

    lcf.var.init(cf, args[1]);                         //初始化，“编译”脚本

    NgxHttpCoreModule::handler(                       //注册处理函数
        cf, &handler_type::handler);
    return NGX_CONF_OK;
}
```

11.6.4 测试验证

在配置文件里修改 `ndg_echo` 指令，改为含有变量的脚本：

```
location /echo {  
    #注意指令后的参数，字符串里有多个变量  
    ndg_echo "hello nginx $current_method from $server_addr\n";  
}
```

执行测试命令“curl http://localhost/echo”，输出的结果是：

```
hello nginx GET from 127.0.0.1
```

可以看到程序确实解析了脚本，输出了变量\$current_method和\$server_addr。

11.7 总结

本章介绍了 Nginx 的变量机制，这是 Nginx 区别于其他 Web 服务器的一大特色。

如果把模块比作 Nginx 宫殿里的房间，那么配置指令就是房间的门窗，可以高效地设置模块的各种参数，但指令创建的成本比较高，而且沟通的方向是单向的。变量是另一种模块与外界通信的方式，它像是房间的水管和电线，可以在房间的任何地方布设任意数量，以很随意的方式与外部沟通交流，输入输出多种信息，模块之间也可以使用变量互相交换信息，用法非常灵活。

Nginx 变量机制的核心是两个结构体： ngx_variable_value_t 和 ngx_http_variable_t。前者表示变量值，是个简单的字符串；后者表示变量访问对象，使用 get/set 函数获取变量值。

Nginx 在 ngx_http_core_module 里存储所有的变量对象，而在每个请求里存放变量值，这种做法分离了变量和变量的表示，能够依据具体的请求来计算相应的变量值。模块要在配置解析阶段使用函数 ngx_http_add_variable() 注册变量，然后在处理 HTTP 请求时调用函数 ngx_http_get_variable() 来获取变量值。^①

为了更有效地利用变量，Nginx 又实现了“脚本”功能，可以在字符串里使用任意多个变量，组合成复杂的文本，这被称为复杂变量。ngx_http_complex_value_t 用于存储复杂变量对象，ngx_http_compile_complex_value_t 则是一个辅助对象，用于在配置解析时“编译”脚本。

本章实现了几个变量的 C++封装类，如 NgxVariables、NgxVarManager 和 NgxComplexValue，可以很容易地完成变量的注册和读写工作。

^① Nginx 里还提供 ngx_http_get_variable_index() 和 ngx_http_get_indexed_variable() 等函数访问变量值，但使用上不如 ngx_http_get_variable() 方便，故本书未介绍。

第 12 章

Nginx辅助设施

本章将介绍一些 Web 开发的辅助技术，包括 MD5/SHA1 摘要、URI 编解码、正则表达式、共享内存等，它们是开发 Web 服务器不可或缺的实用工具。

有很多第三方库可以实现这些功能，但 Nginx 非常体贴地在核心框架之外集成了它们的实现或封装，让我们不必费心费力地再去网络上搜索。当然，如果对 Nginx 内置的实现不满意，我们还可以改用其他的实现，例如 C++ 标准库或者 Boost 程序库。

12.1 摘要算法

摘要算法 (digest, 又称为散列算法、哈希算法)，是一种单向不可逆函数，能够把任意长度的数据“压缩”为一个固定大小的“摘要”字符串，而这个“摘要”是唯一的，可以看作是原数据的“指纹”。^①

摘要算法被广泛应用于密码学、身份鉴定、网络金融等领域，在 Web 服务器里则通常用于数据标识、完整性校验和防篡改。

Nginx 提供了当前较为流行的三种摘要算法：MD5、SHA-1 和 MurmurHash2。

12.1.1 MD5

MD5 (Message Digest) 应该是在程序员群体里最为人知的摘要算法，它能够生成 128 位 (16 字节) 的摘要，运行速度快，性能稳定，适用于各种要求不高的场合。

^① 与现实中的指纹一样，实际上摘要算法也存在“碰撞”的可能，只是几率极小，目前 MD5、SHA-1 算法均已经找到有效的攻击方法，但在低安全性要求的领域仍可使用。

Nginx 在头文件 `<ngx_md5.h>` 里提供对 MD5 算法的支持。注意：它不包含在 `<ngx_core.h>` 里，我们必须单独包含它。

接口函数

MD5 算法使用的结构体和函数如下：

```
typedef ... ngx_md5_t;           //计算 MD5 的结构体定义

void ngx_md5_init(ngx_md5_t *ctx);
void ngx_md5_update(ngx_md5_t *ctx, const void *data, size_t size);
void ngx_md5_final(u_char result[16], ngx_md5_t *ctx);
```

用法

计算 MD5 摘要值需要先定义一个 `ngx_md5_t` 变量，用函数 `ngx_md5_init()` 初始化，然后调用 `ngx_md5_update()` 输入待摘要的数据，最后从 `ngx_md5_final()` 里得到 16 字节的摘要结果。

示范 MD5 用法的代码如下：

```
ngx_md5_t      md5;           //计算 MD5 所需的结构体
u_char        buf[16];       //MD5 值输出，长度是 16 字节

ngx_md5_init(&md5);          //初始化结构体
ngx_md5_update(&md5, "metroid", 7); //输入数据
ngx_md5_final(buf, &md5);     //输出 MD5 值
```

12.1.2 SHA-1

SHA-1 (Secure Hash Algorithm) 是另一种摘要算法，它生成 160 位 (20 字节) 的摘要，安全性要好于 MD5 算法。^①

Nginx 在头文件 `<ngx_sha1.h>` 里提供对 SHA-1 算法的支持，同样的，我们必须单独包含它。

接口函数

自 1.11.2 开始，Nginx 的 SHA-1 算法不再依赖 OpenSSL，完全自行实现，结构体和函数的形式与 MD5 算法基本相同：

① 2017 年初，Google 公布了两份内容不同但 SHA-1 值完全相同的 PDF 文件，意味着 SHA-1 算法被正式攻破。

```
typedef ...    ngx_shal_t;                //计算 SHA1 的结构体定义

void ngx_shal_init(ngx_shal_t *ctx);
void ngx_shal_update(ngx_shal_t *ctx, const void *data, size_t size);
void ngx_shal_final(u_char result[20], ngx_shal_t *ctx);
```

用法

计算 SHA-1 摘要值的步骤也与 MD5 相同，但最后的摘要结果是 20 字节，示范代码如下：

```
ngx_shal_t      sha;                //计算 SHA1 所需的结构体
u_char          buf[20];            //SHA1 值输出，长度是 20 字节

ngx_shal_init(&sha);                //初始化结构体
ngx_shal_update(&sha, "prime", 5);  //输入数据
ngx_shal_final(buf, &sha);          //输出 SHA1 值
```

12.1.3 MurmurHash

MurmurHash 是摘要算法家族里的新成员（发明于 2008 年），对文本字符串有很好的效果，具有更好的随机性。

Nginx 在头文件<ngx_murmurhash.h>里提供了 MurmurHash2 算法^①，接口非常简单：

```
uint32_t ngx_murmur_hash2(u_char *data, size_t len);
```

MurmurHash 算法与 MD5、SHA1 不同，计算不需要额外的结构体，直接调用 ngx_murmur_hash2() 函数就可以得到一个 32 位（4 字节）的数字摘要值。

12.1.4 C++封装

MD5、SHA-1 和 MurmurHash 这三种摘要算法的接口都比较简单，获取摘要值的操作也很流程化，但 C++封装仍然可以得到更易用的接口。

由于 MD5 和 SHA-1 的使用方法类似，所以我们可以利用模板元编程，把结构体和函数作为元数据传递给类，执行摘要流程。而对于 MurmurHash，可以利用模板特化技术，特化出一个专门的类。

摘要算法元数据

元数据类集中定义了三种摘要算法的结构体和操作函数，它们都是编译期的常量：

① 要注意一点，Nginx 实现的 MurmurHash2 算法初始种子被硬编码为 0 值，不能通过参数设置。

```

struct NgxMd5Meta //MD5 算法元数据
{
    typedef ngx_md5_t ctx_type; //计算 MD5 所需的结构体

    static constexpr int len = 16; //MD5 摘要的长度

    static constexpr auto init_func = &ngx_md5_init;
    static constexpr auto update_func = &ngx_md5_update;
    static constexpr auto final_func = &ngx_md5_final;
};

struct NgxShalMeta //SHA1 算法元数据
{
    typedef ngx_shal_t ctx_type; //计算 SHA1 所需的结构体

    static constexpr int len = 20; //SHA1 摘要的长度

    static constexpr auto init_func = &ngx_shal_init;
    static constexpr auto update_func = &ngx_shal_update;
    static constexpr auto final_func = &ngx_shal_final;
};

struct NgxMurmurHash2Meta //MurmurHash2 算法元数据
{
    static constexpr auto update_func = &ngx_murmur_hash2;
};

```

摘要算法类

摘要算法类 `NgxDigest` 从模板参数 `DigestMeta` 里得到所需的结构体类型和函数指针，计算 MD5 和 SHA-1 摘要，流程是完全相同的：

```

template<typename DigestMeta> //模板参数是摘要算法元数据
class NgxDigest final
{
public:
    typedef DigestMeta meta; //简化类型定义
public:
    NgxDigest()
    {
        init(); //构造函数里初始化结构体
    }
    ~NgxDigest() = default;
public:
    void init() //初始化
    {
        meta::init_func(&m_ctx); //调用元函数里的函数指针
    }
};

```

```

void update(const void* data, std::size_t len) //计算摘要值
{
    meta::update_func(&m_ctx, data, len); //调用元函数里的函数指针
}

const u_char* final() //结束摘要
{
    meta::final_func(m_buf, &m_ctx); //调用元函数里的函数指针

    return m_buf; //返回摘要值
}

private:
    typename meta::ctx_type m_ctx; //计算摘要所需的结构体

    u_char m_buf[meta::len]; //摘要值
    u_char m_hex[meta::len*2]; //摘要值的字符串表示
};

```

使用 Nginx 的 ngx_hex_dump() 函数, 可以把二进制数组转换为十六进制的字符串数组 (即 Hex 编码):

```

ngx_str_t str() //返回摘要的字符串表示
{
    ngx_hex_dump(m_hex, m_buf, sizeof(m_buf)); //hex 编码

    return ngx_str_t{sizeof(m_hex), m_hex}; //返回 Nginx 字符串
}

```

我们还可以重载操作符 operator(), 把 NgxDigest 变成一个函数对象, 一次性完成摘要值的计算工作:

```

public:
    ngx_str_t operator()(const void* data, std::size_t len)
    {
        init(); //初始化
        update(data, len); //计算摘要值
        final(); //结束摘要

        return str(); //返回摘要的字符串表示
    }
}

```

特化摘要算法类

MurmurHash 算法的执行流程与 MD5 和 SHA-1 不同, 所以我们使用 C++ 的模板特化技术, 生成一个特别版的 NgxDigest:


```

template<> //模板特化
class NgxDigest<NgxMurmurHash2Meta> final
{
public:
    typedef NgxMurmurHash2Meta meta; //简化类型定义
public:
    NgxDigest() = default;
    ~NgxDigest() = default;
public:
    uint32_t operator()(const void* data, std::size_t len)
    {
        return meta::update_func((u_char*)data, len); //调用元函数里的函数指针
    }
};

```

用法

使用 typedef 可以简化模板类 NgxDigest，给出易用的名字：

```

typedef NgxDigest<NgxMd5Meta>          NgxMd5;
typedef NgxDigest<NgxShalMeta>         NgxShal;
typedef NgxDigest<NgxMurmurHash2Meta>  NgxMurmurHash2;

```

这些摘要类可以像函数一样直接调用：

```

NgxMd5          md5;
NgxShal         shal;
NgxMurmurHash2  murmur2;

cout << md5("abcd",4) << endl;
cout << shal("abcd",4) << endl;
cout << murmur2("abcd",4) << endl;

```

12.2 编码和解码

HTTP 协议是一种文本协议^①，为了支持特殊字符和二进制数据，在处理过程中会有各种编码和解码操作，Nginx 对此提供了较为完善的支持。

12.2.1 CRC 校验

CRC (Cyclic Redundancy Code) 从一段数据产生出校验和，用于检验数据的完整

^① 这里说的是常见的 HTTP 1.1 的情况，HTTP 2.0 已经是二进制字节流了。

性^①，历史相当悠久。CRC 有多个版本，Nginx 实现了 CRC16 和 CRC32，本书只介绍 CRC32。

Nginx 在头文件 `ngx_crc32.h` 里定义了两个计算 CRC32 的函数，声明是：

```
uint32_t ngx_crc32_short(u_char *p, size_t len);
uint32_t ngx_crc32_long(u_char *p, size_t len);
```

这两个函数的计算结果是相同的，区别仅在于内部实现：`ngx_crc32_short()` 使用了较小的查找表，处理短字符串（30~60）速度较快，而 `ngx_crc32_long()` 在处理长字符串时更有优势。

Nginx 也可以分段计算 CRC32，相当于分解执行了 `ngx_crc32_long()` 函数：

```
#define ngx_crc32_init(crc)
void ngx_crc32_update(uint32_t *crc, u_char *p, size_t len);
#define ngx_crc32_final(crc)
```

很多读者应该都已经熟知 CRC 算法，而且 CRC 的计算也比较简单，所以这里不再列出示例代码。

12.2.2 Base64 编码解码

Base64 与 Hex 编码类似，是一种把二进制数据转换为可见字符的方法。它使用 64 个字符^②，编码后的数据长度大约是原长度的 4/3。

标准的 Base64 编码使用了字符 `'/'` 和 `'+'`，而这两个字符会被 URI 编码为 `%2F` 和 `%2B` 的形式，所以 Nginx 也提供支持 URI 的 Base64 编码，字符改用 `'-'` 和 `'_'`。

Nginx 里的 Base64 编码解码函数声明如下：

```
//计算编码或解码后的数据长度
#define ngx_base64_encoded_length(len) (((len + 2) / 3) * 4)
#define ngx_base64_decoded_length(len) (((len + 3) / 4) * 3)

//标准 Base64 编解码
void ngx_encode_base64(ngx_str_t *dst, ngx_str_t *src);
ngx_int_t ngx_decode_base64(ngx_str_t *dst, ngx_str_t *src);

//适用于 URL 的 Base64 编解码
void ngx_encode_base64url(ngx_str_t *dst, ngx_str_t *src);
```

① 某种程度上来说，CRC 也可以算作一种摘要算法，只不过它的算法非常简单，值不唯一，所以用途明确，只是错误校验。

② 相应的，Hex 编码可以称为 Base16。

```
ngx_int_t ngx_decode_base64url(ngx_str_t *dst, ngx_str_t *src);
```

这两组函数除编码字符集的区别外还有一处不同, `ngx_encode_base64()` 使用 '=' 填补结果字符串, 使长度为 4 的整数倍, 而 `ngx_encode_base64url()` 则不会。

示范 Base64 编码的代码如下:

```
ngx_str_t src = ngx_string("mario");           //源字符串
ngx_str_t dst;                                   //目标字符串

dst.len = ngx_base64_encoded_length(src.len);   //计算目标长度
dst.data = NgxPool(r).nalloc<u_char>(dst.len); //分配内存

ngx_encode_base64(&dst, &src);                  //标准 Base64 编码
cout << dst << endl;                            //输出 bWFyaW8=
ngx_encode_base64url(&dst, &src);                //URL Base64 编码
cout << dst << endl;                            //输出 bWFyaW8
```

12.2.3 URI 编码解码

URI 只允许使用有限的字符集, 如果含有特殊字符或者中文则必须要编码, Nginx 为此提供了两个函数:

```
uintptr_t ngx_escape_uri(u_char *dst, u_char *src, size_t size,
                          ngx_uint_t type);
void ngx_unescape_uri(u_char **dst, u_char **src, size_t size,
                      ngx_uint_t type);
```

如果 `ngx_escape_uri()` 的 `dst` 参数是空指针, 那么它会返回需要编码的字符数量 n , 所以目标字符串的长度就应该是原长度 $+2*n$ 。

`type` 参数可以指定编解码的类型:

```
#define NGX_ESCAPE_URI           0           //URI 编码字符集
#define NGX_ESCAPE_ARGS         1           //URI 参数编码字符集
#define NGX_ESCAPE_URI_COMPONENT 2
#define NGX_ESCAPE_HTML         3
#define NGX_ESCAPE_REFRESH      4
#define NGX_ESCAPE_MEMCACHED    5           //memcached 编码字符集
#define NGX_ESCAPE_MAIL_AUTH    6           //邮件编码字符集

#define NGX_UNESCAPE_URI        1           //不处理? 后的参数
#define NGX_UNESCAPE_REDIRECT   2           //只处理不可见字符
```

常用的编码类型是 `NGX_ESCAPE_URI`，使用 URI 的编码集编码。解码时类型 0（Nginx 没有对 0 做宏定义）表示解码整个字符串，否则不处理 “?” 后的参数。

URI 编解码的示范代码如下：

```
ngx_str_t src = ngx_string("test&/echo?name=#link"); //一个 URI
ngx_str_t dst;

dst.len = src.len+ //计算编码后长度
    2*ngx_escape_uri(nullptr,src.data, src.len, NGX_ESCAPE_URI);
dst.data = NgxPool(r).nalloc<u_char>(dst.len); //分配内存

ngx_escape_uri(dst.data, src.data, src.len, NGX_ESCAPE_URI);
cout << dst << endl;
```

程序的输出是：

```
test&/echo%3Fname=%23link #注意 "&" 没有被编码
```

12.2.4 HTML 和 JSON 编码

HTML 和 JSON 也是 Web 服务器经常处理的数据格式，Nginx 也提供了它们的编码操作：

```
uintptr_t ngx_escape_html(u_char *dst, u_char *src, size_t size);
uintptr_t ngx_escape_json(u_char *dst, u_char *src, size_t size);
```

如果 `dst` 参数是空指针，那么函数会返回结果字符串增加的长度。

HTML 编解码的示范代码如下：

```
ngx_str_t html = ngx_string("<html>"); //html 文本
ngx_str_t out;

out.len = html.len+ //计算编码后长度
    ngx_escape_html(nullptr, html.data, html.len);
out.data = NgxPool(r).nalloc<u_char>(out.len); //分配内存

ngx_escape_html(out.data, html.data, html.len);
cout << out << endl;
```

输出结果是：

```
&lt;html&gt; # "<>" 已经被编码
```

12.3 正则表达式

正则表达式是文本处理领域里的一个强大工具，它定义了一套完善的语法规则，可以执行匹配、查找、提取、验证等许多操作，在 Web 服务器里处理 HTTP 相关文本时非常有用。

Nginx 自身不具备正则表达式功能，而是利用了著名的 PCRE 库，相关的数据结构和函数均定义在头文件 `<ngx_regex.h>` 里。

PCRE 库功能强大，速度也足够快，但本书认为更应该使用 C++ 标准库里的 `std::regex` 或者 Boost 的 `xpressive` 库，它们同样提供了完善且快速的正则表达式功能，而且接口更加清晰易用。

本节简略介绍功能更丰富的 Boost.xpressive 库。

Boost.xpressive 库使用类 `cregex/sregex` 定义正则表达式，使用类 `cmatch/smacth` 保存正则匹配的结果。功能函数有匹配的 `regex_match()`、查找的 `regex_search()` 和替换的 `regex_replace()` 等。

下面的代码示范了 Boost.xpressive 库的部分用法：

```
using namespace boost::xpressive;                                // 打开名字空间

cmatch what;                                                    // 存储匹配的结果
auto reg = cregex::compile("(.*?):(.*?)");                      // 一个正则表达式

auto rc = regex_match("host: 127.0.0.1", what, reg);           // 匹配正则表达式
assert(rc);                                                      // 断言匹配成功
cout << what[1] << " " << what[2] << endl;                    // 输出匹配的子表达式

auto reg2 = cregex::compile(R"((\w+)= (\w+))");                // 另一个正则表达式

regex_search("a=1&b=2", what, reg2);                            // 正则查找
cout << what[1] << " " << what[2] << endl;                    // 输出匹配的子表达式
```

Boost.xpressive 库还支持静态正则表达式，可以在编译期实现正则表达式，运行效率更高，甚至可以自定义小型语法解析器，读者可以进一步阅读推荐书目 [3] 了解更多的用法。

12.4 共享内存

共享内存是操作系统里一种高效的进程间通信方式，也是 Nginx 里 worker 进程间共享数据的一种便捷手段，Nginx 封装了 `mmap`、`shmget` 等系统调用，可以在多种操作系统上支持共享内存特性。

12.4.1 结构定义

Nginx 使用 `ngx_shm_t` 结构表示一块共享内存:

```
//定义在 os/unix/ngx_shmem.h
typedef struct {
    u_char*      addr;           //共享内存的地址
    size_t       size;          //共享内存的大小
    ngx_str_t     name;          //共享内存的名字
    ngx_log_t*    log;           //日志对象
    ngx_uint_t    exists;        //是否存在的标志位
} ngx_shm_t;
```

还有一个 `ngx_shm_zone_t` 结构, 专门用来管理共享内存块:

```
//定义在 core/nginx_cycle.h
typedef struct ngx_shm_zone_s ngx_shm_zone_t;
typedef ngx_int_t (*ngx_shm_zone_init_pt) (ngx_shm_zone_t *zone, void *data);

struct ngx_shm_zone_s {
    void*          data;          //初始化函数的回调参数
    ngx_shm_t      shm;          //共享内存结构体
    ngx_shm_zone_init_pt init;    //初始化函数指针
    void*          tag;          //共享内存关联的标记
};
```

Nginx 会在配置解析阶段把所有的 `ngx_shm_zone_t` 结构放入 `ngx_list_t` 链表, 集中创建共享内存。

12.4.2 操作函数

我们可以越过 Nginx 框架, 直接创建和释放共享内存:

```
ngx_int_t    ngx_shm_alloc(ngx_shm_t *shm);    //创建共享内存
void         ngx_shm_free(ngx_shm_t *shm);     //释放共享内存
```

但 Nginx 并不推荐这种方式, 而是建议使用函数 `ngx_shared_memory_add()`, 在配置解析阶段向链表里添加 `ngx_shm_zone_t` 对象, 由 Nginx 在 `ngx_init_cycle()` 里统一为模块创建共享内存。

`ngx_shared_memory_add()` 的声明如下:

```
ngx_shm_zone_t *ngx_shared_memory_add(ngx_conf_t *cf, ngx_str_t *name,
                                       size_t size, void *tag);
```

12.4.3 C++共享内存

虽然使用 Nginx 的 `ngx_shm_t` 结构和相关函数可以很容易地创建共享内存，但它的使用却是一大难题，因为我们必须自行管理内存区域（分配、回收、整理），还需要使用原子量或者互斥锁解决多进程并发操作的问题，Nginx 对此并没有提供更方便的接口。

在 C++ 里可以使用 `boost.interprocess` 库，它实现了可移植的进程间通信功能，接口简单易用，其中就包括了共享内存操作。

`boost.interprocess` 库是一个头文件库，不需要编译即可使用，但它的一些底层系统调用需要使用 `librt`，所以 Nginx 在 `configure` 时要使用 `--with-ld-opt="-lrt"` 参数，或者在模块的 `config` 脚本里使用变量 `ngx_module_libs`。

在 `boost.interprocess` 库里可以使用类 `managed_shared_memory` 创建一块共享内存，然后就能在里面创建任意的对象，如原子量、字符串甚至 `vector`、`map` 等标准容器，所有的 Nginx worker 进程都可以共享使用。

下面的代码简单示范了在共享内存里原子量的用法，统计 Nginx 发送的总字节数，更详细的代码可参考 GitHub 资源：

```
//简化类型定义
typedef NgxLogDebug                                     log;
typedef boost::interprocess::managed_shared_memory    shmем_type;
typedef boost::atomic<long>                           atomic_type;

auto& cf = this_module::instance().conf().main(r);      //获取配置数据

shmем_type segment(                                     //打开或创建共享内存
    boost::interprocess::open_or_create, "ndg_shmem",  //使用一个名字
    cf.size);                                           //还要提供大小

auto& counter =                                         //使用原子变量
    *segment.find_or_construct<atomic_type>("counter")(0);

counter += r->connection->sent;                        //累加发送字节数

log(r).print("sent=%l", static_cast<long>(counter));  //记录日志
```

需要注意的是共享内存属于系统级资源，不会因为 Nginx 进程结束而自动消失，所以我们需要在 Nginx 退出时销毁共享内存，也就是使用模块的 `exit_master` 回调函数，例如：

```
static void exit_master(ngx_cycle_t *cycle)
{
```

```

//退出 master 进程时销毁共享内存
boost::interprocess::shared_memory_object::remove("ndg_shmem");
}

static ngx_module_t m =                                     //模块定义
{
    NGX_MODULE_V1,
    ctx(),                                                    //函数指针表
    cmds(),                                                    //配置指令数组
    NGX_HTTP_MODULE,                                           //HTTP 模块标志
    NGX_MODULE_NULL(6),                                       //其他回调
    &exit_master,                                              //退出 master 进程回调
    NGX_MODULE_V1_PADDING
};

```

boost.interprocess 库不仅提供了共享内存，还有互斥量、条件变量、信号量、文件锁、消息队列等很多现代操作系统中常见的进程间通信机制，都可用于 Nginx 模块开发，读者可以参考 Boost 库的文档以了解更多用法。

12.5 总结

本章简要介绍了四个在 Web 服务器开发中比较常用的技术，是 Nginx 宫殿外锦上添花的装饰。

摘要算法可以为数据生成独一无二的标识，用于完整性校验和防篡改。Nginx 内置了 MD5、SHA-1 和 MurmurHash2 三种算法，在内部的文件缓存和安全连接模块使用了 MD5 算法。

TCP/HTTP 传输中经常要对数据做编码解码操作，Nginx 提供了 CRC、Base16、Base64 和 URI/HTML/JSON 编解码等函数。

正则表达式是处理文本的强有力工具，Nginx 利用 PCRE 库实现了对正则表达式的支持。PCRE 库由 C 语言实现，小巧快速，但学习和使用的成本较高。

共享内存是进程间通信的一种高效方式，Nginx 封装了操作系统的 API，可以便捷地创建共享内存。但 Nginx 没有提供操作共享内存的高级接口，需要程序员自行管理内存，还要使用锁机制处理并发访问。

这些技术与 Nginx 框架的联系都不是很密切，所以完全可以按照自己的喜好选择 Nginx 之外的实现。例如可以改用 std::regex 或者 boost::xpressive 来书写正则表达式，或者使用 boost.interprocess 库以 C++类的方式操作共享内存。

第 13 章

Nginx 进程机制

从这里开始，我们将逐步深入 Nginx 系统内部，关注 `core`、`os/unix` 等目录，从源码的级别来解析 Nginx 的工作机制，揭开 Nginx 高性能、高稳定性的秘密。

Nginx 通常都运行在类 UNIX 操作系统上（最常见的就是 Linux），使用 `one master/multi workers` 的进程模式运行，`master` 进程负责管理监控，而 `worker` 进程则负责提供对外的网络服务。这种方式不仅充分利用了 CPU，也很好保证了服务的稳定。

本章将从 UNIX 系统调用入手，剖析 Nginx 的进程相关数据结构和启动流程，详细研究 `single`、`master` 和 `worker` 进程，它们是 Nginx 整个运行机制的起点。

13.1 基本系统调用

作为一个成熟完善的操作系统，UNIX 提供了众多的系统调用函数，它们是系统内核的入口，想要高效地利用系统资源就必须充分熟悉这些系统调用。^①

已经存在很多详尽介绍这些 UNIX 系统调用的网络资料或者书籍了，所以本章仅是做一个简明扼要的说明，不涉及实现原理、用法细节和注意事项，读者可以参考推荐书目 [6] 了解相关的详细信息，但更方便的是直接使用 UNIX 系统提供的在线参考手册，也就是“`man 2 xxx`”。

13.1.1 `errno`

UNIX 系统调用会返回一个 `int` 值表示函数的执行结果，失败通常是 -1，而表示具体原

^① 通常所说的“系统调用”是内核接口和库函数的统称，并不十分准确，本书中的系统调用指的是操作系统内核对外暴露的函数接口。

因的错误代码则保存在另外一个全局变量里，这就是 `errno`。^①

`errno` 的类型是标准的 `int`，含义不够清楚，所以 Nginx 做了重命名：

```
// 位于 os/unix/nginx_errno.h
typedef int          ngx_err_t;           // 重命名错误码的类型
#define ngx_errno    errno               // 获取系统调用错误码
#define ngx_set_errno(err)  errno = err   // 设置系统调用错误码
```

为了保持对各种操作系统的兼容，Nginx 也重定义了错误代码，例如：

```
#define NGX_EPERM      EPERM              // 无权限访问
#define NGX_ESRCH      ESRCH              // 进程不存在
#define NGX_EINTR      EINTR              // 发生系统中断
#define NGX_ECHILD     ECHILD             // 无子进程
#define NGX_ENFILE     ENFILE             // 系统文件描述符不足
#define NGX_EMFILE     EMFILE             // 系统文件描述符不足
```

13.1.2 getrlimit

计算机系统里的资源都是有限的，包括 CPU 时间、进程数量、栈空间大小、文件数量等，不能无限制地使用，函数 `getrlimit()` 可以获取这些信息：

```
int getrlimit(int resource, struct rlimit *rlim); // 获取系统资源限制
```

不过在 Nginx 里只使用了两个：`RLIMIT_CORE` 和 `RLIMIT_NOFILE`，前者用于获取 `core` 文件上限，而后者获取最大文件描述符数量，也就是最多能够打开的 `socket` 数量。

13.2 进程系统调用

进程 (process) 是静态程序的动态实体，是 UNIX 系统里的核心概念，操作系统以进程为单位分配 CPU、内存、文件描述符等资源，调度管理进程是操作系统的最基本职责。

13.2.1 getpid

在 UNIX 系统里，任何进程都有一个作为标识的唯一 ID 号，也就是 `pid`，可以用 `getpid()` 函数获取，它的声明是：

```
pid_t getpid(void); // 获取当前进程的 pid
```

① 在不同的系统里，`errno` 可能不是一个真正的变量，而是一个宏，真正的实现也许是函数。它不是系统调用，所以位于参考手册的第 3 部分。

为了屏蔽操作系统差异，Nginx 使用宏进行了重命名：

```
// 位于 os/unix/nginx_process.h
typedef pid_t          ngx_pid_t;          //重命名进程 ID 类型

#define NGX_INVALID_PID  -1                //使用-1 表示无效的 pid
#define ngx_getpid        getpid           //重命名 getpid 系统调用
```

13.2.2 fork

函数 fork() 可以创建一个新的进程，原进程称为父进程，新进程称为子进程。

子进程精确地复制了父进程的代码段和所有其他数据，包括堆、栈、工作目录、打开的文件描述符等，但并不会发生代价高昂的内存拷贝，而采用的是“写时复制”(copy on write)技术，只有当父进程或子进程修改数据（写操作）时才会发生真正的页拷贝。

fork() 的声明如下：

```
pid_t fork(void);                                //创建新的进程
```

由于函数执行后产生了新的进程，它的返回结果在父进程和子进程里也就有了差异：

- 在父进程里，函数执行成功后返回子进程的 pid；
- 在父进程里，函数执行失败返回-1，错误原因可从 errno 获得；
- 在子进程里，函数执行成功后返回 0。

在编写程序时需要使用返回值来判断父进程或子进程，从而执行不同的后续流程。

fork() 函数的执行如图 13-1 所示。

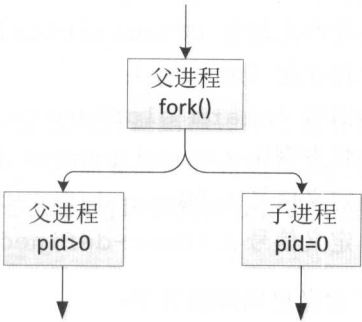


图 13-1 fork() 函数的执行

13.2.3 waitpid

在 fork 之后两个进程都会独立运行，但父进程通常还需要对子进程负责，当子进程结束

时处理其返回信息，否则子进程就成了所谓的“僵尸进程”(zombie)。

函数 `waitpid()` 可以等待一个子进程结束，并得到它的结束状态信息：

```
pid_t waitpid(pid_t pid, int *status, int options);    //等待子进程结束
```

参数 `pid` 可以明确指定等待某个子进程，值为 `-1` 就会等待任意一个子进程。为了避免等待造成的阻塞，参数 `options` 可以设置为 `WNOHANG`。

如果有子进程已经结束，那么函数返回子进程的 `pid`，并在 `status` 里存储子进程的结束状态，否则返回 `0`。

13.3 信号系统调用

信号 (signal) 是 UNIX 操作系统里进程通信的一种重要手段，进程依靠信号来获取通知，从而改变自身的状态以致行为。

13.3.1 kill

向进程发送信号需要使用函数 `kill()`，声明是：^①

```
int kill(pid_t pid, int sig);    //向进程发送信号
```

Nginx 里主要用到的信号有：

- `SIGKILL` : 强制进程停止运行 (Kill, unblockable);
- `SIGQUIT` : 要求进程正常退出 (Quit);
- `SIGINT` : 要求进程中断 (Interrupt);
- `SIGTERM` : 要求进程终止运行 (Termination);
- `SIGHUP` : 要求进程挂起 (Hungup);
- `SIGALRM` : 定时器信号 (Alarm clock);
- `SIGCHLD` : 子进程状态变化 (Child status has changed), 通常是结束;
- `SIGUSR1` : 用户自定义信号 1 (User-defined signal 1);
- `SIGUSR2` : 用户自定义信号 2 (User-defined signal 2)。

Nginx 对部分信号重定义了含义更明确的名字：

```
// 位于 core/nginx_config.h
#define NGX_SHUTDOWN_SIGNAL    QUIT    //正常结束运行
```

^① `kill()` 系统调用位于参考手册的第 2 部分，直接使用“man kill”会看到 Shell 命令 `kill`。

```
#define NGX_TERMINATE_SIGNAL    TERM           //立即结束运行
#define NGX_RECONFIGURE_SIGNAL HUP            //重新加载配置文件
#define NGX_REOPEN_SIGNAL      USR1           //重新打开文件
#define NGX_CHANGEBIN_SIGNAL   USR2           //更新可执行文件，热升级
```

13.3.2 sigaction

在进程中处理信号需要使用函数 `sigaction()`：^①

```
int sigaction(int sigum,                //设置信号的处理函数
              const struct sigaction *act, struct sigaction *oldact);
```

它为信号 `sigum` 指定了 `act->sa_handler` 作为信号处理函数，结构体 `sigaction` 的声明是：

```
struct sigaction {
    void      (*sa_handler)(int);        //信号的处理函数
    ...
};                                       //其他成员变量
```

当进程收到信号时就会执行 `act->sa_handler`，因为信号是实时的，所以处理函数应该尽量简单，不要有过于复杂的操作，避免阻塞进程，稍后我们会看到 Nginx 是如何处理的。

13.3.3 sigsuspend

函数 `sigsuspend()` 设置信号掩码 (`mask`) 并挂起进程，直至收到信号才唤醒进程：

```
int sigsuspend(const sigset_t *mask);    //挂起进程，收到信号才唤醒
```

如果一直没有收到信号，那么进程将持续在函数调用处挂起（休眠），不占用 CPU。

13.4 结构定义

Nginx 的进程机制涉及三个重要的数据结构，本节将详细介绍 `ngx_cycle_t`、`ngx_core_conf_t` 和 `ngx_process_t`。需要注意，它们与 Nginx 的模块架构的耦合程度是很低的，可以说是基本互不干扰，这也是 Nginx 精妙设计的体现。

13.4.1 ngx_cycle_t

我们之前已经在第 6 章初步接触了 `ngx_cycle_t`，它是 Nginx 框架里最核心的数据结构，包含了 Nginx 运行时所需的所有关键信息，本节只摘录与进程相关的部分代码：

^① 传统的处理信号函数是 `signal()`，但它可移植性很差，不推荐使用。

```

// 位于 core/nginx_core.h
typedef struct ngx_cycle_s          ngx_cycle_t;

// 位于 core/nginx_cycle.h
struct ngx_cycle_s {
    void****          conf_ctx;          //配置数据的起始存储位置

    ngx_pool_t*       pool;              //基本的内存池
    ngx_log_t*        log;              //基本的日志对象

    ngx_module_t**    modules;          //运行时模块指针数组, 所有的模块都在这里

    ngx_array_t       listening;        //监听端口数组
    ngx_connection_t* connections;      //连接池
    ngx_event_t*      read_events;      //读事件池, 与连接对应
    ngx_event_t*      write_events;     //写事件池, 与连接对应

    ngx_array_t       paths;            //使用的目录
    ngx_list_t        open_files;       //使用的文件
    ngx_list_t        shared_memory;    //使用的共享内存

    ngx_cycle_t*      old_cycle;        //“上一个” Nginx 的 cycle 对象

    ngx_str_t         conf_file;        //启动时的配置文件
    ngx_str_t         conf_param;       //启动时的-g 参数
    ngx_str_t         conf_prefix;     //启动时的配置文件目录
    ngx_str_t         prefix;          //启动时的工作目录
    ngx_str_t         hostname;        //当前主机名
};

```

可以看到, `ngx_cycle_t` 里的成员都是 Nginx 运行时所必需的重要数据, 例如配置信息、模块数组、监听端口、连接池、共享内存等, 每个后面都是一个很大的子系统。

一个 Nginx 进程有且仅有一个在用的 `ngx_cycle_t` 对象, 为了保证在程序的任何位置都能够访问这个核心对象, Nginx 使用了一个全局指针 `ngx_cycle`:

```

// 位于 core/nginx_cycle.c
volatile ngx_cycle_t*  ngx_cycle;      //指示当前使用的 ngx_cycle_t 对象

```

注意它被特别声明为 `volatile`, 提醒编译器此变量可能随时被改变, 不能做存储优化, 这是因为在 reload 或 update binary 时 Nginx 会利用指针来切换新旧 cycle 对象。

13.4.2 ngx_core_conf_t

`ngx_core_conf_t` 是第 0 号模块 `ngx_core_module` 的配置结构体, 包含了 Nginx

运行最基本的参数，在 6.3.1 节我们有过基本了解，它与 Nginx 进程相关的代码摘要如下：

```
typedef struct {
    ngx_flag_t      daemon;           //守护进程标志量
    ngx_flag_t      master;           //master 进程标志量

    ngx_msec_t      timer_resolution; //时间精确度，单位是毫秒
    ngx_msec_t      shutdown_timeout; //worker 进程的超时退出时间

    ngx_int_t        worker_processes; //worker 进程的数量

    ngx_int_t        rlimit_nofile;    //可打开的最大文件数量
    off_t            rlimit_core;      //coredump 文件大小

    char*            username;         //运行使用的用户名
    ngx_uid_t        user;             //运行使用的 uid
    ngx_gid_t        group;            //运行使用的 gid

    ngx_str_t        working_directory; //工作目录，用于存放 coredump

    ngx_str_t        pid;              //master 进程的 pid 文件名
    ngx_str_t        oldpid;           //new binary 时旧的 pid 文件名
} ngx_core_conf_t;
```

ngx_core_conf_t 只是一个简单的运行时参数记录，大多数成员都可以与配置文件里的指令对应。

13.4.3 ngx_process_t

在 UNIX 系统里，父进程并不会自动保存子进程的相关信息，所以 Nginx 定义了 ngx_process_t 结构，用来记录每个 worker 子进程的基本信息：

// 位于 os/unix/nginx_process.h

```
typedef struct {
    ngx_pid_t        pid;              //子进程的 pid
    int              status;           //子进程的结束状态，即 waitpid() 的输出

    ngx_spawn_proc_pt proc;            //子进程执行的函数
    void*            data;             //proc 函数的参数
    char*            name;             //子进程的名字 (title)

    //以下标志位是子进程目前的状态
    unsigned          respawn:1;       //重新生成的新子进程
    unsigned          just_spawn:1;    //子进程刚刚产生
    unsigned          detached:1;      //子进程已经与父进程分离
```

```

    unsigned    exiting:1;    //子进程正在退出
    unsigned    exited:1;    //子进程已经退出
} ngx_process_t;

```

成员 `proc` 是产生子进程时需要执行的函数，它的声明是：

```
typedef void (*ngx_spawn_proc_pt) (ngx_cycle_t *cycle, void *data);
```

当某个子进程异常退出时，Nginx 会使用 `proc`、`data` 等参数“原地满血复活”子进程。

Nginx 使用一个全局数组保存所有产生的子进程，这就是“进程池”：

```

// 位于 os/unix/nginx_process.c
#define          NGX_MAX_PROCESSES          1024          //最多 1024 个子进程
ngx_process_t   ngx_processes[NGX_MAX_PROCESSES];        //子进程信息数组

```

`ngx_processes` 数组是静态分配的，不能动态增长，所以 Nginx 只支持生成最多 1024 个子进程（也就是 worker 进程），这远大于目前主流计算机的 CPU 核心数量，足够用了。

管理 `ngx_processes` 进程数组还需要使用另外两个全局变量：

```

// 位于 os/unix/nginx_process.c
ngx_int_t       ngx_process_slot;          //刚创建的子进程索引号
ngx_int_t       ngx_last_process;          //标记数组的最后一个空位

```

13.5 全局变量

Nginx 大量使用了全局变量用于在各个函数之间通信，而且分散在多个源文件里，有些杂乱，上一节我们已经见到了几个，本节再对这些全局变量做一个较详细的归纳整理。

13.5.1 命令行相关

Nginx 使用函数 `ngx_get_options()` 解析命令行参数，解析的结果存储在下面的这些变量里：

```

// 位于 core/nginx.c
static ngx_uint_t   ngx_show_help;          //-h/-?参数，显示帮助信息

static ngx_uint_t   ngx_show_version;        //-v 参数，显示版本信息
static ngx_uint_t   ngx_show_configure;      //-V 参数，显示编译配置信息

static u_char*      ngx_prefix;              //-p 参数，工作路径
static u_char*      ngx_conf_file;           //-c 参数，配置文件
static u_char*      ngx_conf_params;         //-g 参数，环境信息
static char*        ngx_signal;              //-s 参数，信号字符串

```



```
// 位于 core/nginx_cycle.c
ngx_uint_t      ngx_test_config;      // -t 参数, 测试配置文件
ngx_uint_t      ngx_dump_config;      // -T 参数, dump 整个配置文件
ngx_uint_t      ngx_quiet_mode;        // -q 参数, 安静模式, 不输出测试信息
```

注意前几个 `ngx_show_help`、`ngx_show_version` 等变量是 `static` 全局的, 所以只在 `nginx.c` 里可访问, 供 `main()` 等使用; 而 `ngx_test_config` 等变量是真正全局生效的, 不仅 `ngx_cycle.c` 里的函数可以访问, `nginx.c` 里的 `main()` 等函数也可以使用。

Nginx 还完全保存了命令行参数, 用于设置进程名和热升级启动新进程:

```
// 位于 os/unix/nginx_process.c
int      ngx_argc;      // 命令行参数数量
char**   ngx_argv;      // 原始命令行参数数组的拷贝
char**   ngx_os_argv;   // 原始命令行参数数组
```

13.5.2 操作系统相关

Nginx 使用三个全局变量记录了操作系统的信息:

```
// 位于 os/unix/nginx_posix_init.c
ngx_int_t      ngx_ncpu;      // 当前系统里的逻辑 CPU 数量
ngx_int_t      ngx_max_sockets; // 可打开的描述符数量, 目前暂未使用
ngx_os_io_t     ngx_os_io;    // UNIX 系统基本的数据收发函数集合
```

`ngx_os_io` 里有 6 个函数指针, 是 Nginx 收发数据的关键, 将在第 14 章详细讲解。

13.5.3 进程功能相关

当前进程 (master/workers) 的基本数据保存在下面的全局变量里:

```
// 位于 os/unix/nginx_process_cycle.c
ngx_pid_t      ngx_pid;      // 当前进程的 pid
ngx_uint_t     ngx_worker;    // 当前进程在 Nginx 内部的 id 号
ngx_uint_t     ngx_process;   // 当前进程的类型
```

`ngx_pid` 是进程的 `pid`, 由 UNIX 操作系统分配, 而 `ngx_worker` 则是 Nginx 内部为进程分配的有序整数, 程序可以使用它来更明确地标识不同的进程。^①

`ngx_process` 标记了当前进程的类型, 取值是:

^① 变量 `ngx_worker` 只有在 Nginx 1.9.1 之后的版本里才可用。

```
// 位于 os/unix/nginx_process_cycle.h
#define NGX_PROCESS_SINGLE 0 //single 进程, 单一进程
#define NGX_PROCESS_MASTER 1 //master 进程, 管理进程
#define NGX_PROCESS_SIGNALLER 2 //signal 进程, 只发送信号的进程
#define NGX_PROCESS_WORKER 3 //worker 进程, 工作进程
#define NGX_PROCESS_HELPER 4 //辅助进程
```

还有一个全局变量标记 Nginx 是否启用了守护进程模式:

```
// 位于 os/unix/nginx_process_cycle.c
ngx_uint_t ngx_daemonized; //守护进程标志量
```

13.5.4 信号功能相关

为了使信号处理函数尽量简单, Nginx 为可处理的信号都设置了对应的全局变量, 收到信号时只要执行一个简单的赋值操作, 使信号处理的阻塞几乎降到了零。

这些信号全局变量是:

```
// 位于 os/unix/nginx_process_cycle.c
sig_atomic_t ngx_reap; //SIGCHLD, 子进程状态变化, 需要重新产生
sig_atomic_t ngx_sigalrm; //SIGALRM, 更新时间的信号
sig_atomic_t ngx_terminate; //SIGTERM, 终止进程
sig_atomic_t ngx_quit; //SIGQUIT, 正常退出
sig_atomic_t ngx_reconfigure; //SIGHUP, 重新加载配置文件, 也就是 reload
sig_atomic_t ngx_reopen; //SIGUSR1, 重新打开所有文件
sig_atomic_t ngx_change_binary; //SIGUSR2, 更新可执行文件
```

因为进程通常都有很多连接要处理, 在收到终止信号到所有连接均完全关闭时会有一段时间, 所以需要用一个变量标记“正在退出”的状态:

```
ngx_uint_t ngx_exiting; //退出过程中, 进程正在关闭连接
```

13.6 启动过程

所有的 C 程序都要从 main() 函数开始执行, Nginx 当然也不例外, 本节简要分析启动时 main() 的主要流程, 它是 Nginx 进程机制的“点火钥匙”。

13.6.1 基本流程

main() 位于源文件 core/nginx.c, 工作流程可以简略描述如下:

- 1) 解析命令行参数, 显示帮助信息;

- 2) 初始化时间、日志、内存池、正则表达式等基本功能;
- 3) 根据命令行参数建立一个基本的 cycle 对象;
- 4) 初始化静态模块数组 ngx_modules;
- 5) 核心操作, 创建进程使用的真正 cycle, 里面有解析配置文件、启动监听端口等;
- 6) 如果使用了 -s 参数, 那么发送信号后退出;
- 7) 如果配置了 “daemon on”, 那么守护进程化;
- 8) 把进程的 pid 写入 pid 文件, 用于将来的发送信号;
- 9) 根据配置进入单进程 (single) 或多进程 (master/worker) 模式;
- 10) 无论是哪种模式, 进程内部都是用无限循环来处理事件, 只有当收到停止信号或者发生异常时才会退出循环, main() 函数才会真正结束。

在这些步骤中重点是第 5 步和第 9 步, 我们会在后面的小节里详细阐述。

13.6.2 解析命令行

Nginx 首先使用函数 ngx_get_options() 解析命令行参数:

```
if (ngx_get_options(argc, argv) != NGX_OK) {    //解析命令行参数
    return 1;                                    //解析失败返回 1 直接退出
}
```

Nginx 并没有使用标准的库函数 getopt(), 而是完全自行实现了命令行参数的解析。解析成功后, 命令行的字符串参数就转换成了 13.5.1 节里的那些全局变量。

如果使用了 -s 参数, 那么会额外设置全局变量 ngx_process 为 NGX_PROCESS_SIGNALLER, 表示进程将要发送信号, 后续的流程中就不会进入事件处理循环:

```
if (ngx_strcmp(ngx_signal, "stop") == 0        //目前-s支持的4种信号字符串
    || ngx_strcmp(ngx_signal, "quit") == 0
    || ngx_strcmp(ngx_signal, "reopen") == 0
    || ngx_strcmp(ngx_signal, "reload") == 0)
{
    ngx_process = NGX_PROCESS_SIGNALLER;        //设置进程类型, 只发送信号
}
```

13.6.3 版本和帮助信息

处理“-h/-?/-v/-V”很简单，只需要检查全局变量，输出预先准备好的帮助信息：

```
if (ngx_show_version) {                                //使用了-h/-?/-v/-V
    ngx_show_version_info();                            //显示版本和帮助信息

    if (!ngx_test_config) {                             //如果没有-t/-T
        return 0;                                       //那么显示帮助信息结束，退出
    }
}
```

13.6.4 初始化 cycle

ngx_cycle_t 对象（之后简称为 cycle 对象）保存了 Nginx 运行时的重要数据，数据的来源包括命令行参数、环境变量、配置文件，甚至还有之前运行的实例，所以必须综合处理各种信息，这方面 Nginx 考虑的非常周到。

临时 cycle 对象

Nginx 首先创建一个临时的 cycle 对象，为之后真正使用的 cycle 对象做准备：

```
ngx_pid = ngx_getpid();                                //获取进程的 pid
log = ngx_log_init(ngx_prefix);                        //初始化 log 对象

ngx_memzero(&init_cycle, sizeof(ngx_cycle_t));        //初始化临时 cycle 对象
init_cycle.log = log;                                  //初始化 cycle 的 log
```

随后是一个关键操作，把全局变量指针 ngx_cycle 指向这个临时对象，这样之后的操作就不必再关心具体使用的是哪个 cycle，而是使用指针指向的对象：

```
ngx_cycle = &init_cycle;                             //重要操作，初始化 cycle 全局指针
```

有了 ngx_cycle 指针后，Nginx 继续初始化其他的重要参数，注意这时还是操作的临时 cycle 对象：

```
ngx_save_argv(&init_cycle, argc, argv);                //保存命令行参数到全局变量
ngx_process_options(&init_cycle);                      //初始化配置文件名、运行目录
ngx_os_init(log) != NGX_OK;                            //初始化操作系统信息
ngx_crc32_table_init();                                //初始化 crc32 计算表
ngx_preinit_modules();                                 //初始化模块数组
```

这些操作完成之后，Nginx 就获得了以下的基本信息：

- 可用的日志对象，即 init_cycle.log;

- 命令行参数的拷贝，在全局变量 `ngx_argc`、`ngx_argv` 等里（见 13.5.1 节）；
- 配置文件名、工作目录等，在 `init_cycle.conf_file/prefix` 等里；
- 操作系统的 CPU 数、收发函数，在 `ngx_ncpu`、`ngx_os_io` 等里（见 13.5.2 节）；
- 所有的静态模块，在 `ngx_modules` 数组里（见第 6 章）。

创建 cycle 对象

以临时对象 `init_cycle` 为样板，Nginx 调用函数 `ngx_init_cycle()` 开始创建真正的 `cycle` 对象：^①

```
cycle = ngx_init_cycle(&init_cycle);           //创建真正的 cycle 对象

ngx_init_cycle() 首先把前面获得的所有基本信息拷贝到新建的 cycle 里：

pool = ngx_create_pool(NGX_CYCLE_POOL_SIZE, log); //创建一个新的内存池
cycle = ngx_palloc(pool, sizeof(ngx_cycle_t)); //在新内存池里创建 cycle 对象

cycle->pool = pool;                             //使用新的内存池
cycle->log = log;                                //使用原日志对象
cycle->old_cycle = old_cycle;                     //保存临时 cycle 对象备用

//从临时 cycle 对象拷贝 conf_prefix、prefix、conf_file 等配置文件字符串
cycle->conf_prefix.len = old_cycle->conf_prefix.len;
cycle->conf_prefix.data = ngx_pstrdup(pool, &old_cycle->conf_prefix);

//创建配置结构体数组，大小是总模块数量
cycle->conf_ctx = ngx_palloc(pool, ngx_max_module * sizeof(void *));

ngx_cycle_modules(cycle);                       //拷贝静态模块数组到本 cycle
```

这时 `cycle` 对象已经持有了配置文件和模块数组，并且创建了配置结构体数组，模块架构已经就绪，可以开始配置解析工作了。这部分的功能已经在本书的第 6 章做了详细的介绍，这里不再重复，请读者及时复习。

配置文件解析完，Nginx 就获得了定义在配置文件里的更多信息，例如工作模式、进程数量、动态模块、共享内存、监听端口等，`cycle` 的初始化即将进入尾声。

接下来 Nginx 依据 `cycle` 里的信息做运行前最后的准备工作，包括创建目录、打开文件、创建共享内存、打开监听端口、调用 `init_module` 初始化模块等，这些都会被之后 `fork` 出

^① `ngx_init_cycle()` 有近 1000 行，函数实现代码过长，也许将来 Nginx 会重构为若干个小函数，更容易理解和维护。

的子进程共享，代码虽然很多但基本上都是操作数组、链表等数据结构，并不难理解，为节约篇幅在此不列出具体的代码。

新 cycle 对象创建完成之后，临时对象 `init_cycle` 就不需要再使用了，所以 `ngx_cycle` 指针就指向函数返回的刚创建好的 `cycle`：

```
ngx_cycle = cycle; //重要操作，初始化 cycle 全局指针
```

在后续的流程里 Nginx 将一直使用这个 `cycle` 对象。

13.6.5 测试配置

`cycle` 对象初始化时配置文件解析成功意味着配置正确，如果命令行里有“-t/-T”参数要测试配置，那么就在这里输出测试的结果，然后退出：

```
if (ngx_test_config) { //使用-t/-T，要求测试配置
    if (!ngx_quiet_mode) { //不是安静模式
        ... //输出一条测试成功的信息
    }

    if (ngx_dump_config) { //使用-T，要求 dump 配置文件内容
        ... //遍历配置文件，输出到 stdout
    }
    return 0; //测试配置结束，退出
}
```

13.6.6 发送信号

如果使用了“-s”要求发送信号，那么进程的类型就是 `NGX_PROCESS_SIGNALLER`，需要给正在运行的其他 Nginx 进程发送信号，而不是要监听端口处理连接，Nginx 就会执行函数 `ngx_signal_process()`，然后退出：

```
if (ngx_signal) { // -s 发送 quit、stop 等信号
    return ngx_signal_process(cycle, ngx_signal); //需要向所有子进程发送信号
} //发送信号后进程结束
```

信号的具体处理过程可参见 13.7 节。

13.6.7 守护进程化

Nginx 检查 `core` 模块的配置 `ngx_core_conf_t` 里的 `daemon` 成员，如果要求守护进程化就执行 `ngx_daemon()` 函数：

```

if (!ngx_inherited && ccf->daemon) {           //要求守护进程化
    if (ngx_daemon(cycle->log) != NGX_OK) {     //执行 ngx_daemon() 函数
        return 1;
    }
    ngx_daemonized = 1;                         //守护进程化完毕，设置标记
}

```

虽然 UNIX 提供了库函数 `daemon()`，但 Nginx 还是自行实现了 `ngx_daemon()` 函数，逻辑基本相同，都使用 `fork()` 然后关闭标准输入输出：

```

// 位于 os/unix/nginx_daemon.c
ngx_int_t ngx_daemon(ngx_log_t *log)          //守护进程化
{
    switch (fork()) {                          //fork 调用，产生子进程
        case -1:                               //fork 调用出错
            return NGX_ERROR;
        case 0:                                //返回 0，是子进程
            break;                             //继续后面的流程
        default:                               //返回 pid，是父进程
            exit(0);                           //父进程直接结束运行
    }

    ngx_pid = ngx_getpid();                    //这里是子进程执行的代码
    ...                                         //关闭标准输入输出
    return NGX_OK;                             //守护进程化完毕
}

```

`ngx_daemon()` 与 `daemon()` 只有一点不同，那就是调用了 `getpid()` 给 `ngx_pid` 赋值，这是因为之前的 `ngx_pid` 是父进程的，而在 `fork()` 之后父进程已经消失了（调用了函数 `exit`），由子进程继续之后的处理，所以必须重新获取正确的 `pid`。

13.6.8 启动工作进程

全局变量 `ngx_process` 表示的是 Nginx 进程的类型，它的初始值是 0，也就是 `NGX_PROCESS_SINGLE`，即 Nginx 默认是单进程模式。

如果配置文件里使用了指令 “`master_process on`”，那么进程的类型就会改为 `NGX_PROCESS_MASTER`：

```

if (ccf->master && ngx_process == NGX_PROCESS_SINGLE) {
    ngx_process = NGX_PROCESS_MASTER;          //改为 master 进程模式
}

```

在 `main()` 函数的最后，Nginx 依据 `ngx_process` 决定使用单进程还是多进程模式：

```
if (ngx_process == NGX_PROCESS_SINGLE) { //要求单进程模式
    ngx_single_process_cycle(cycle); //进入单进程生命周期循环
} else { //否则就是多进程模式
    ngx_master_process_cycle(cycle); //进入 master 生命周期，并产生 worker 子进程
}
```

单进程模式和多进程模式的具体工作流程见后面的 13.8 和 13.9 节。

13.6.9 流程图

Nginx 在 main() 函数里的启动及运行过程如图 13-2 所示。

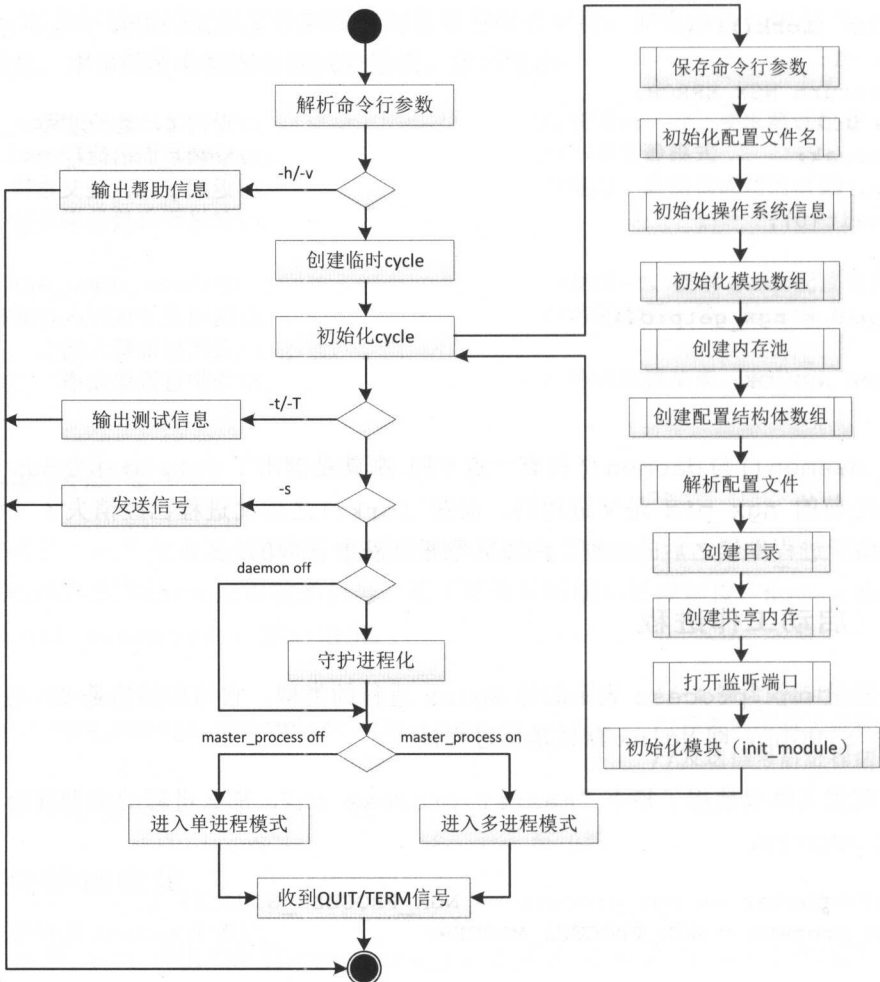


图 13-2 Nginx 在 main() 函数里的启动及运行过程

从流程图里可以看到，只有使用了“-h/-v/-t/-s”等命令行参数启动 Nginx，或者接收到 UNIX 信号时，Nginx 进程才会正常退出，否则将会持续运行，在无限循环里使用事件机制对外提供网络服务。

13.7 信号处理

Nginx 的信号处理与大多数 UNIX 程序类似，都是设置信号对应的处理函数，但它使用了全局变量来作为信号的响应，处理函数的逻辑非常简单，这种做法很值得学习借鉴。

13.7.1 信号处理函数

处理信号的关键是信号与处理函数的对应关系，所以 Nginx 定义了一个数组，实现了这两者之间的映射，并且简化了信号的组织形式：

```
// 位于 os/unix/nginx_process.c
ngx_signal_t signals[] = {
    { ngx_signal_value(NGX_RECONFIGURE_SIGNAL), //SIGHUP 信号
      "SIG" ngx_value(NGX_RECONFIGURE_SIGNAL), //信号的字符串表示
      "reload", //对应的命令行字符串
      ngx_signal_handler }, //对应的处理函数
    ...
    { SIGALRM, "SIGALRM", "", ngx_signal_handler }, //SIGALRM 信号
    { SIGINT, "SIGINT", "", ngx_signal_handler }, //SIGINT 信号
    { SIGCHLD, "SIGCHLD", "", ngx_signal_handler }, //SIGCHLD 信号
    { 0, NULL, "", NULL } //空对象，数组结束
};
```

注意数组里每个 UNIX 信号对应的处理函数都是 ngx_signal_handler()。

在 Nginx 启动时，main() 函数调用 ngx_init_signals() 遍历数组，使用系统调用 sigaction() 指定信号的处理函数：

```
// 位于 os/unix/nginx_process.c
ngx_int_t ngx_init_signals(ngx_log_t *log) //初始化信号处理
{
    for (sig = signals; sig->signo != 0; sig++) { //遍历数组
        ngx_memzero(&sa, sizeof(struct sigaction));
        sa.sa_handler = sig->handler; //填充结构体里的处理函数
        sigemptyset(&sa.sa_mask);
        if (sigaction(sig->signo, &sa, NULL) == -1) { //系统调用设置处理函数
            return NGX_ERROR; //设置失败则启动失败
        }
    }
}
```

```

    }
    return NGX_OK;
}

```

13.7.2 发送信号

如果命令行里使用“-s”要求发送信号，Nginx 就会执行函数 `ngx_signal_process()`，它先从配置结构体的 `ccf->pid` 里获得 pid 文件名，读出 pid——也就是正在运行的 Nginx master 进程，然后调用函数 `ngx_os_signal_process()` 发送信号。

`ngx_os_signal_process()` 的逻辑很简单，由于 `signals` 数组里保存了字符串与信号的映射关系，所以就逐个比较字符串，使用系统调用 `kill()` 向 master 进程发送信号：

```

ngx_int_t
ngx_os_signal_process(ngx_cycle_t *cycle, char *name, ngx_pid_t pid)
{
    for (sig = signals; sig->signo != 0; sig++) {        //遍历信号数组
        if (ngx_strcmp(name, sig->name) == 0) {          //比较字符串
            if (kill(pid, sig->signo) != -1) {            //kill 向 pid 发送信号
                return 0;                                  //成功返回 0
            }
        }
    }
    return 1;                                              //没找到或发送失败返回 1
}

```

13.7.3 处理信号

虽然 Nginx 可以处理很多的信号，但它们都使用同一个处理函数 `ngx_signal_handler()`，它只是使用 `switch-case` 简单地设置信号对应的全局变量（13.5.4 节），代码摘要如下：

```

for (sig = signals; sig->signo != 0; sig++) { //遍历信号数组
    if (sig->signo == signo) {                 //找到对应的信号，即可处理
        break;
    }
}

case NGX_PROCESS_MASTER:                      //master/single 进程处理信号
case NGX_PROCESS_SINGLE:
    switch (signo) {
        case ngx_signal_value(NGX_SHUTDOWN_SIGNAL): //SIGQUIT
            ngx_quit = 1;                             //设置变量 ngx_quit
            break;
        case ngx_signal_value(NGX_TERMINATE_SIGNAL): //SIGTERM
        case SIGINT:                                   //SIGINT

```

```

    ngx_terminate = 1;                //设置变量 ngx_terminate
    break;
case SIGALRM:                        //SIGALRM
    ngx_sigalrm = 1;                  //设置变量 ngx_sigalrm
    break;
case SIGCHLD:                        //SIGCHLD
    ngx_reap = 1;                    //设置变量 ngx_reap
    break;
}
break;                                //master/single 进程处理信号结束

case NGX_PROCESS_WORKER:            //worker/helper 进程处理信号
case NGX_PROCESS_HELPER:
...
break;                                //worker/helper 进程处理信号结束
}

```

master 进程与 worker 进程对信号的处理不完全相同，这是因为 master 进程负责管理工作，必须接受更多的信号，而 worker 进程职责相对简单，只需要处理 SIGQUIT、SIGTERM 等少量的信号。

对于子进程发生变化的信号 SIGCHLD，master 进程还要维护进程数组 ngx_processes，使用系统调用 waitpid() 获取子进程的结束状态：

```

if (signo == SIGCHLD) {              //子进程状态有变化，通常是意外退出
    ngx_process_get_status();         //维护进程数组，获取子进程的结束状态
}

```

13.8 单进程模式

在单进程模式下，Nginx 的核心功能函数是 ngx_single_process_cycle()，它集成了多进程模式里 master 进程和 worker 进程的所有功能，但因为只有一个进程，不需要管理子进程，也没有进程间同步的麻烦，实现上反而比较简单。

13.8.1 single 进程

ngx_single_process_cycle() 的主要工作流程如下：

- 调用所有模块的 init_process 函数指针，执行模块的进程初始化；
- 进入无限循环 for(;;)；
- 调用 ngx_process_events_and_timers() 处理连接，这是 Nginx 处理网络连

接与定时器的主业务逻辑，将在第 14 章详细讲解；

- 处理信号，如果要求退出则调用所有模块的 `exit_process` 函数指针，然后结束。

`ngx_single_process_cycle()` 的代码摘要如下：

```
for (i = 0; cycle->modules[i]; i++) {
    if (cycle->modules[i]->init_process) {           //模块的进程初始化操作
        if (cycle->modules[i]->init_process(cycle) == NGX_ERROR) {
            exit(2);                                //初始化失败则直接退出
        }
    }
}

for ( ;; ) {                                       //无限循环开始
    ngx_process_events_and_timers(cycle);           //处理各种网络事件和定时器事件

    if (ngx_terminate || ngx_quit) {               //SIGTERM 和 SIGQUIT 信号
        for (i = 0; cycle->modules[i]; i++) {
            if (cycle->modules[i]->exit_process) { //模块的进程退出操作
                cycle->modules[i]->exit_process(cycle);
            }
        }
        ngx_master_process_exit(cycle);             //删除 pid，模块清理，关闭监听端口
    }

    if (ngx_reconfigure) {                         //SIGHUP 信号
        ngx_reconfigure = 0;                        //清除标志量，防止重复进入
        cycle = ngx_init_cycle(cycle);              //使用当前 cycle 重新初始化
        ngx_cycle = cycle;                          //使用新的 cycle 对象
    }

    if (ngx_reopen) {                              //SIGUSR1 信号
        ngx_reopen = 0;                             //清除标志量，防止重复进入
        ngx_reopen_files(cycle, (ngx_uid_t) -1);    //重新打开文件
    }
}                                                    //无限循环代码结束
```

在无限 `for` 循环里的函数 `ngx_process_events_and_timers()` 是 Nginx 的功能主体，它使用 `epoll/kqueue` 等事件机制来处理并发的海量网络连接事件，但这不是本章关注的重点，读者可参见第 14 章了解它的细节。

处理网络事件之外，`for` 循环的工作就是检查信号对应的全局变量，如果为 1 就表示收到了某个信号，必须及时处理。由于单进程模式通常用于调试和验证，不用于真正的生产环境，所以它的处理也就略微“粗暴”一点。

对于 SIGTERM 和 SIGQUIT 信号, Nginx 不做区分统一处理, 不考虑现有连接是否还在收发数据, 不设置 ngx_exiting 状态, 直接关闭所有的监听端口, 最后调用 exit(0) 结束。

对于 SIGHUP 信号, Nginx 把当前的 cycle 对象 (也就是 ngx_cycle 全局指针) 作为参数传递给 ngx_init_cycle(), 以它为模板创建出新的 cycle 对象, 这个新 cycle 对象完全复制了当前 cycle 的命令行参数、工作目录, 但却更新了配置文件, 随后 ngx_cycle 全局指针就切换到新 cycle 对象, 很简单地完成了 reload。

对于 SIGUSR1 信号, Nginx 直接调用 ngx_reopen_files(), 遍历文件链表 cycle->open_files, 重新打开这些文件。

13.8.2 single 进程流程图

single 进程的工作流程如图 13-3 所示。

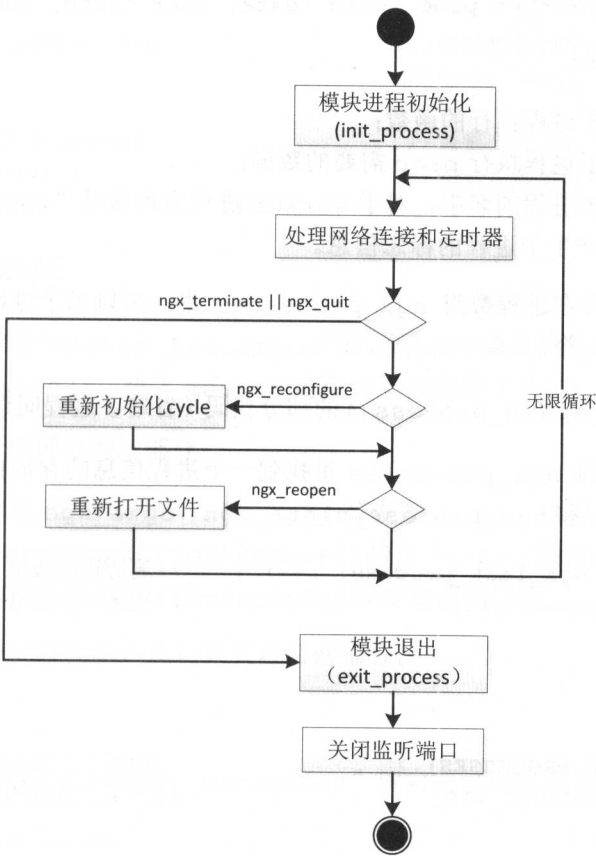


图 13-3 single 进程的工作流程

13.9 多进程模式

Nginx 的多进程模式要比单进程模式复杂，不是把 single 进程直接拆开那么简单，主要是因为存在多个进程，需要用信号维护它们之间的关系。

13.9.1 产生子进程

因为 master 进程必须在 ngx_processes 数组里维护子进程的状态，所以产生子进程不能像 ngx_daemon() 那样简单地调用 fork(), Nginx 专门用一个函数 ngx_spawn_process() 来产生子进程，声明如下：

```
typedef void (*ngx_spawn_proc_pt) (ngx_cycle_t *cycle, void *data);

ngx_pid_t ngx_spawn_process(ngx_cycle_t *cycle,
    ngx_spawn_proc_pt proc, void *data, char *name, ngx_int_t respawn);
```

函数参数的意义是：

- proc : 子进程执行的函数；
- data : 子进程执行 proc 需要的数据；
- name : 子进程的名字，对于 worker 进程来说就是 “worker process”；
- respawn: 产生子进程的标志信息。

这些参数都被保存在进程数组 ngx_processes 里，所以当子进程意外退出时就可以很容易地原样 “再生” 一个。

下面摘录了 ngx_spawn_process() 的部分代码（省略了进程间通信的 socketpair）。

Nginx 先在进程池 ngx_processes 里找到一个进程信息的存放位置，使用了全局变量 ngx_last_process 和 ngx_process_slot：

```
for (s = 0; s < ngx_last_process; s++) {    //遍历进程数组
    if (ngx_processes[s].pid == -1) {        //找到一个没有使用的空位
        break;
    }
}

if (s == NGX_MAX_PROCESSES) {                //超过了 1024 的限制
    return NGX_INVALID_PID;                  //产生子进程失败
}

ngx_process_slot = s;                        //设置全局变量，当前使用的数组位置
```

之后调用 `fork()` 产生子进程，这是本函数的关键：

```
pid = fork();                                //fork 系统调用产生子进程

switch (pid) {                               //pid 判断父子进程，分支处理
case -1:                                     //产生子进程失败
    return NGX_INVALID_PID;
case 0:                                     //子进程分支
    ngx_pid = ngx_getpid();                 //重新获取 pid，即子进程的 pid
    proc(cycle, data);                     //执行子进程的功能，无限循环
    break;
default:                                    //父进程分支
    break;                                  //继续后面的程序
}
```

`fork()` 之后，子进程——也就是 `worker` 进程就执行了 `proc(cycle, data)`，正式进入工作模式——无限循环直至收到信号 `exit`，而父进程则维护子进程的信息：

```
ngx_processes[s].pid = pid;                 //在进程数组里记录子进程的信息
...                                          //name、data、respawn 等其他信息

if (s == ngx_last_process) {               //调整变量 ngx_last_process
    ngx_last_process++;                     //即进程数组的长度
}
```

13.9.2 master 进程

当配置文件里指定了“`master process on`”时，`main()` 函数的最后就会执行函数 `ngx_master_process_cycle()`，进入 `master` 进程的工作流程（见 13.6.8 节）。

`master` 进程的工作可分为三部分：

- 使用命令行参数 `ngx_argv`、`ngx_argv` 等设置 `master` 进程的名字（title）；
- 调用 `ngx_start_worker_processes()` 启动 `worker` 进程；^①
- 无限循环，使用系统调用 `sigsuspend()` 等待并处理信号。

设置进程名的代码比较简单，我们只关注后两部分。

启动 worker 进程

`master` 进程根据指令“`worker_processes`”确定的进程数量创建子进程：

^① 本书暂不考虑 `cache` 进程。

```
ngx_start_worker_processes(cycle, ccf->worker_processes, ...);
```

函数 `ngx_start_worker_processes()` 启动 worker 子进程, 实现代码如下 (省略了 channel 部分):

```
static void
ngx_start_worker_processes(ngx_cycle_t *cycle, ngx_int_t n, ngx_int_t type)
{
    ngx_int_t i; //产生 worker 进程的计数器

    for (i = 0; i < n; i++) { //循环 n 次, 产生 n 个子进程
        ngx_spawn_process(cycle, ngx_worker_process_cycle,
            (void *) (intptr_t) i, "worker process", type);
    }
}
```

这段代码虽然很短, 但却非常重要, 在调用 `ngx_spawn_process()` 时传入了几个重要的参数, 决定了 worker 子进程的行为:

- `ngx_worker_process_cycle`, 即 worker 进程的功能函数, 内部是无限循环;
- `i`, worker 进程的 id, 转换为了 `void*` 供 `ngx_worker_process_cycle` 使用;
- “worker process”, worker 进程的名字。

`ngx_start_worker_processes()` 执行之后, 就产生了 `n` 个名字是 “worker process” 的子进程, 即 worker 进程, 每个子进程都在执行 `ngx_worker_process_cycle()` 里的无限循环处理网络事件和定时器事件——也就是对外提供 Web 服务, 而 master 进程作为父进程则用进程数组 `ngx_processes` 保存了这些子进程的 pid 等信息。

循环处理信号

启动 worker 进程之后, master 进程进入自己的无限循环 `for(;;)`, 核心操作是系统调用 `sigsuspend()`, 暂时挂起进程, 不占用 CPU, 只有收到信号时才被唤醒, 检查 `ngx_signal_handler()` 所设置的全局变量, 决定具体的进程管理行为:

```
live = 1; //子进程的存活标志
for ( ;; ) { //进入无限循环, 只处理信号
    sigsuspend(&set); //挂起等待信号
    ... //处理各种信号
}
```

for 循环内的代码与单进程模式的 `ngx_single_process_cycle()` 很相似, 但增加了 “善后” 的处理, 用变量 `live` 标记是否还有 worker 进程在运行, 如果是 `live`, 就表示 worker 进程还没有处理完剩余的连接, 就不能立即终止运行。

SIGCHLD 信号（子进程终止运行）

收到 SIGCHLD 信号，意味着有子进程状态发生了变化，最大的可能就是异常退出，真正的原因已经由函数 `ngx_signal_handler()` 保存在了进程数组 `ngx_processes` 里，所以 Nginx 调用 `ngx_reap_children()` 找到被意外结束的进程，使用 `proc/data/name` 等参数重新产生子进程：

```
if (ngx_reap) {
    ngx_reap = 0;
    live = ngx_reap_children(cycle);
}
```

//可能有子进程异常退出
//清除标志量，防止重复进入
//检查进程数组，重启子进程

通过这种方式，master 进程维护了进程池的稳定性，保证即使有 worker 进程意外崩溃也会迅速恢复，这就是 Nginx 能够稳定运行的秘密所在。

SIGQUIT 信号（-s quit）

收到 SIGQUIT 信号，master 进程不会像 single 进程那样立刻终止运行，而是向所有 worker 进程发送 SIGQUIT，要求它们处理完连接后再停止运行，随后继续用 `sigsuspend()` 等待子进程结束的 SIGCHLD 信号。

```
if (ngx_quit) {
    ngx_signal_worker_processes(cycle,
                                ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
    continue;
}
```

//收到 SIGQUIT 信号
//发送 SIGQUIT 要求终止
//回到循环开头，挂起等待信号

由于变量 `ngx_quit` 不清零，所以 master 进程会一直发送 SIGQUIT，而 `ngx_reap_children()` 不会重启主动退出的子进程，所以最后 `live` 将变成 0，这时就可以安全退出：

```
if (!live && (ngx_terminate || ngx_quit)) { //收到 SIGTERM、SIGQUIT 信号
    ngx_master_process_exit(cycle);
    //没有存活子进程则直接 exit
}
```

早期的 Nginx 在这里存在一个问题，如果某个 worker 进程保持着长连接，或者陷入了死循环，那么 Nginx 就永远无法退出了。不过 Nginx 1.11.11 引入了“`worker_shutdown_timeout`”指令，可以让 worker 进程在一个限定的时间内结束，见 13.9.4 节。

SIGTERM 信号（-s stop）

master 进程对 SIGTERM 信号的处理与 SIGQUIT 类似，但它没有那么多“耐心”，略显“粗暴”，使用了一个计时器等待最多 1000 毫秒，之后不管 worker 进程是否已经完成工作都直接发送 SIGKILL 强制终止：

```

if (ngx_terminate) { //收到 SIGTERM 信号
    if (delay > 1000) { //等待了 1 秒钟后仍然有子进程存活
        ngx_signal_worker_processes(cycle, SIGKILL); //发送 SIGKILL 强制终止
    } else { //不到 1 秒钟, 再等等
        ngx_signal_worker_processes(cycle, //发送 SIGTERM 要求终止
            ngx_signal_value(NGX_TERMINATE_SIGNAL));
    }
    continue; //返回无限循环, 继续检查
}

```

SIGHUP 信号 (-s reload)

SIGHUP 信号的处理与 `ngx_single_process_cycle()` 里的差不多, 首先同样以当前 `cycle` 为模板创建一个重新配置的新 `cycle` 对象, 但因为是多进程模式, 还需要用新 `cycle` 生成一批新的 worker 进程, 然后再关闭旧的 worker 进程:

```

if (ngx_reconfigure) { //收到 SIGHUP 信号
    ngx_reconfigure = 0; //清除标志量, 防止重复进入

    cycle = ngx_init_cycle(cycle); //使用当前 cycle 重新初始化
    ngx_cycle = cycle; //使用新的 cycle 对象
    ngx_start_worker_processes( //使用新配置启动新子进程
        cycle, ccf->worker_processes, NGX_PROCESS_JUST_RESPAWN);

    ngx_msleep(100); //阻塞等待 100 毫秒

    live = 1; //设置子进程存活标志
    ngx_signal_worker_processes(cycle, //停止之前的旧子进程
        ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
}

```

因为向旧子进程发送的是 SIGQUIT, 所以这里也与“-s quit”存在同样的问题, 可能 reload 后在系统里会出现一些名为“worker process is shutting down”的 Nginx “死”进程。以前只能用 kill 强制终止运行, 而在 Nginx 1.11.11 之后则可以用“worker_shutdown_timeout”指令来解决。

SIGUSR2 信号

SIGUSR2 信号意味着更新了 Nginx 的可执行程序, Nginx 会利用系统环境变量“NGINX”传递当前打开的监听端口, 重命名 pid 文件为 `ccf->oldpid`, 使用系统调用 `execve()` 执行新程序:

```

if (ngx_change_binary) { //SIGUSR2 信号
    ngx_change_binary = 0;
}

```

```
ngx_new_binary = ngx_exec_new_binary(cycle, ngx_argv);  
}
```

之后只要用“kill -QUIT”就可以停止旧 master 进程。

13.9.3 master 进程流程图

master 进程里对全局变量的处理比较复杂，结合图 13-4 可以更好地理解。

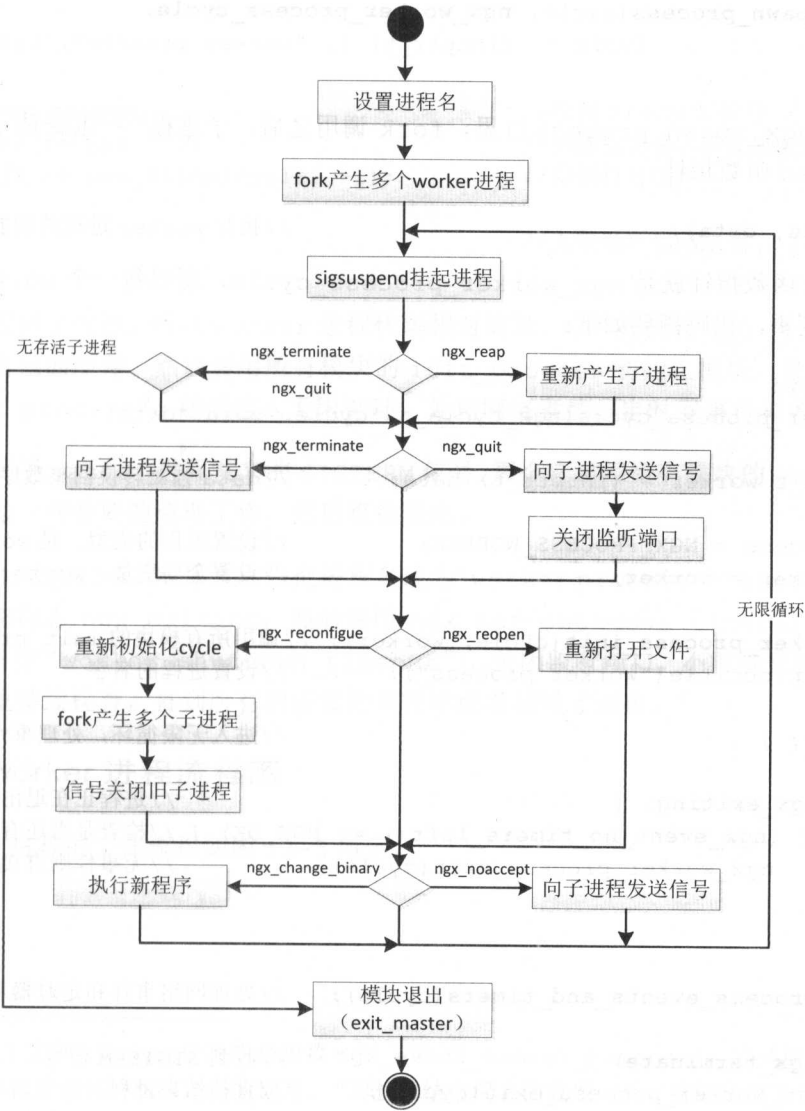


图 13-4 master 进程的工作流程

13.9.4 worker 进程

worker 进程由函数 `ngx_start_worker_processes()` 产生, 由于 `fork` 系统调用的特性, worker 进程完全共享了 master 进程 `fork` 之前的所有数据, 包括工作目录、模块数组、配置文件、共享内存、监听端口等, 这里再列出这段代码 (可参见 13.9.2 节):

```
ngx_int_t i; //产生 worker 进程的计数器
for (i = 0; i < n; i++) { //循环 n 次, 产生 n 个子进程
    ngx_spawn_process(cycle, ngx_worker_process_cycle,
        (void *) (intptr_t) i, "worker process", type);
}
```

在函数 `ngx_spawn_process()` 里, `fork` 调用之后, 子进程——也就是 worker 进程开始执行 `proc` 函数指针:

```
proc(cycle, data); //执行 worker 进程的功能
```

而 `proc` 函数指针就是 `ngx_worker_process_cycle`, 所以每一个 worker 进程都会运行同样的逻辑, 代码摘要如下:

```
static void
ngx_worker_process_cycle(ngx_cycle_t *cycle, void *data)
{
    ngx_int_t worker = (intptr_t) data; //data 强制转换回整数序号

    ngx_process = NGX_PROCESS_WORKER; //设置进程的类型, 是 worker
    ngx_worker = worker; //设置全局变量, worker 进程的序号

    ngx_worker_process_init(cycle, worker); //调用所有模块的 init_process 初始化
    ngx_setproctitle("worker process"); //设置进程的名字

    for ( ;; ) { //进入无限循环, 处理事件和信号

        if (ngx_exiting) { //进程正在退出阶段
            if (ngx_event_no_timers_left() == NGX_OK) { //检查是否还有事件等待处理
                ngx_worker_process_exit(cycle); //无事件则直接结束进程
            }
        }

        ngx_process_events_and_timers(cycle); //处理网络事件和定时器事件

        if (ngx_terminate) { //收到 SIGTERM 信号
            ngx_worker_process_exit(cycle); //直接结束进程
        }
    }
}
```

```

if (ngx_quit) {                                     //收到 SIGQUIT 信号
    ngx_quit = 0;                                   //清除标志量, 防止重复进入
    ngx_setproctitle("worker process is shutting down");

    if (!ngx_exiting) {                             //正在退出则不会执行下面的逻辑
        ngx_exiting = 1;                           //设置正在退出标志
        ngx_set_shutdown_timer(cycle);              //设置超时关闭的时间
        ngx_close_listening_sockets(cycle);         //关闭监听端口
        ngx_close_idle_connections(cycle);          //关闭空闲连接
    }
}

if (ngx_reopen) {                                   //收到 SIGUSR1 信号
    ngx_reopen = 0;                                 //清除标志量, 防止重复进入
    ngx_reopen_files(cycle, -1);                   //重新打开文件
}
                                                    //无限循环代码结束
}

```

因为不管理子进程, 所以 worker 进程代码相对简单, 它首先把传入的 data 强制转换为整数, 作为自己的序号, 然后调用所有模块的 `init_process` 初始化模块, 设置进程的名字为 “worker process”, 随后进入无限循环, 处理网络事件、定时器事件和各种信号。

如果收到了 master 进程发来的 SIGTERM 信号, 那么它调用所有模块的 `exit_process` 函数指针, 做一些必要的清理工作, 然后直接退出。

如果是 SIGQUIT 信号, 那么它先把进程名改为 “worker process is shutting down”, 设置退出状态标志 `ngx_exiting`, 然后调用 `ngx_set_shutdown_timer()` 设置结束的超时时间 (即指令 “worker_shutdown_timeout”), 关闭监听端口, 不再接受新的连接请求, 然后在循环里持续检查, 直到所有的连接处理完毕或者超时才退出。^①

13.9.5 worker 进程流程图

worker 进程的工作流程如图 13-5 所示。

① 在 1.11.11 之前, Nginx 使用的是函数 `ngx_event_cancel_timers()`, 尝试取消定时器事件, 也就是说不再继续等待网络连接的读写, “尽快” 结束现有的收发工作, 这就导致了有可能无法取消所有的定时器事件, 造成进程一直处于 “shutting down” 状态。

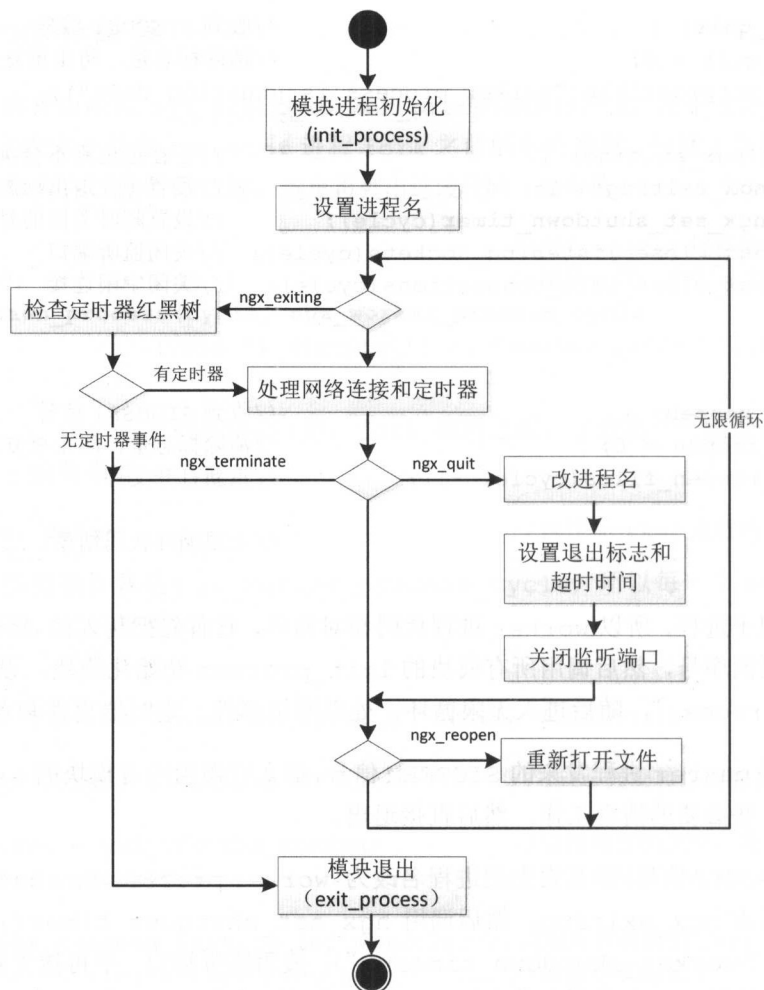


图 13-5 worker 进程的工作流程

13.10 总结

本章介绍 Nginx 的进程机制，它是深埋于 Nginx 宫殿之下的坚实地基，Nginx 的所有功能——模块体系、事件机制、多线程、TCP/UDP/HTTP 处理——都构建在进程机制之上。

`ngx_cycle_t` 包含了 Nginx 运行时的许多重要信息，例如工作目录、配置信息、模块数组、监听端口、连接池、共享内存等，是 Nginx 框架里最核心的数据结构，可以说它就代表了 Nginx 进程。每个 Nginx 进程有且仅有一个在用的 `ngx_cycle_t` 对象，通过全局指针 `ngx_cycle` 来访问。

在启动的时候, Nginx 的工作就是创建 `ngx_cycle` 对象, 为此需要读取命令行参数、环境变量或者配置文件, 初始化 `cycle` 里的各个成员。这其中的一个主要任务是解析配置文件, 由模块架构解析各种指令, 建立起层次化的模块配置信息, 并初始化 `event`、`http`、`stream` 等模块。

`cycle` 对象创建之后, 依据配置 Nginx 就会进入 `single` 进程模式或者 `master/worker` 进程模式, 内部使用无限循环, 通过事件机制的函数 `ngx_process_events_and_timers()` 对外提供服务, 直至收到 `SIGQUIT` 或 `SIGTERM` 信号才终止。

Nginx 对 UNIX 信号的处理很有特色, 使用了若干全局变量来标记信号的有无, 信号处理函数仅设置这些变量的值, 而变量的检查则在无限循环里。这种方式既不会丢失信号, 同时也最小化了信号处理逻辑, 避免了对业务逻辑的阻塞。

当在命令行里使用 “`-s quit/stop/reload/reopen`” 等参数时, Nginx 会读取配置文件里指定的 `pid` 文件, 获得 `master` 进程的 `pid`, 然后向 `master` 进程发送信号 (发送信号的进程随即退出), 再由 `master` 进程向各个 `worker` 进程发送信号。

`master` 进程不处理网络连接, 只关心 UNIX 信号, 使用进程池数组管理所有的 `worker` 进程, 收到信号时就会检查 `worker` 进程的状态, 重启或者终止进程。`worker` 进程与 `master` 进程则正相反, 主要处理网络连接, 不管理其他进程, 被动地接收信号。`single` 进程相当于 `master` 进程和 `worker` 进程的结合体, 但因为只有一个进程, 所以逻辑反而更简单。

Nginx 的进程机制涉及的底层系统知识较多, 实现源码也较分散, 读者应耐心阅读本章的内容, 必要时可使用 `gdb` 跟踪调试 Nginx 的启动过程来加深理解。

第 14 章

Nginx 事件机制

在第 13 章里我们讨论了 Nginx 的进程机制，它让多个 worker 进程分散运行在多个 CPU 上，可以充分利用多核 CPU。但这只是 Nginx 高性能的基础，Nginx 能够高效率处理网络连接的根本上是它的事件处理模型。

所谓的“事件”（event）可以理解作为一种通知机制，当有 I/O 事件发生时系统就会通知进程。

其实一般意义上的“事件”在我们的日常工作生活中很常见，敲打键盘、点击鼠标、触摸屏幕等都是“事件”，操作系统等待并捕获这些“事件”，再发送给相应的进程处理（当然也可以不处理）。

UNIX 信号也可以看作是一种简单的“事件”，但它只有通知而没有额外的数据收发要求，所以大多数情况下不作为“事件”来处理。而在 Web Server 领域，“事件”通常指的是网络事件，即网络连接上的数据可读或者可写，进程对事件处理操作就是及时地接收或者发送数据。

处理事件有很多种方式，Nginx 没有使用常见的多线程，而使用的是“事件驱动”（event driven）机制，在一个单线程里用 `epoll/kqueue` 收集并“消费”各种事件，再配合非阻塞 socket 调用，驱动了整个 Nginx 的运转。

这种方式避免了线程切换的 CPU 损耗，已经被实践证明是简单而高效的，本章就将全面展示 Nginx 的事件机制。

14.1 基本系统调用

本节介绍 UNIX 里与网络编程和定时器相关的几个系统调用。

14.1.1 errno

网络编程使用的 socket 系列函数同样遵循 UNIX 系统调用的规范，用 `errno` 来表示具体的错误原因，但为了与普通的系统调用错误更明确地区分开，Nginx 又做了重命名：

```
// 位于 os/unix/nginx_errno.h
#define ngx_socket_errno      errno          //获取 socket 调用错误码
#define ngx_set_socket_errno(err)  errno = err //设置 socket 调用错误码
```

与 socket 函数相关的常见错误码有：

```
#define NGX_EINPROGRESS      EINPROGRESS      //已发起连接，但还没有成功建立连接
#define NGX_ECONNRESET      ECONNRESET      //连接被复位 (RST)
#define NGX_EADDRINUSE      EADDRINUSE      //地址已被占用 (例如 TIME WAIT)
#define NGX_ECONNREFUSED    ECONNREFUSED    //连接被拒绝
#define NGX_ETIMEDOUT        ETIMEDOUT        //超时错误

#ifdef __hpux__
#define NGX_EAGAIN          EWOULDBLOCK      //非阻塞调用专用错误码，未准备好需重试
#else
#define NGX_EAGAIN          EAGAIN           //非阻塞调用专用错误码，未准备好需重试
#endif
```

14.1.2 ioctl

函数 `ioctl()` 管理系统的 I/O，比起另一个功能类似的函数 `fcntl()` 来说它更简单高效：

```
int ioctl(int d, int request, ...);          //调整设备的属性
```

Nginx 使用它来设置 socket 为非阻塞模式：^①

```
// 位于 os/unix/nginx_socket.c
int ngx_nonblocking(ngx_socket_t s)
{
    int nb = 1;
    return ioctl(s, FIONBIO, &nb);          //设置 socket 为非阻塞模式
}
```

14.1.3 setitimer

函数 `setitimer()` 设置了一个内核级别的定时器，当时间到时操作系统就会向进程发送信号 `SIGALRM`，通常用于进程的计时工作：

① `ioctl` 只需要一次系统调用，而如果操作系统不支持 `ioctl` 那么 Nginx 就会使用 `fcntl`，它需要两次调用才能设置为非阻塞。

```
int setitimer(int which, const struct itimerval *new_value,
              struct itimerval *old_value);
```

时间值由下面的两个结构体决定：

```
struct timeval {
    time_t      tv_sec;           //seconds
    suseconds_t tv_usec;         //microseconds
};

struct itimerval {
    struct timeval it_interval;   //next value
    struct timeval it_value;      //current value
};
```

注意 timeval 的成员 tv_usec，名字是“usec”，表示的是微秒而不是毫秒。

14.1.4 gettimeofday

函数 gettimeofday() 可以获取系统当前的时间，精度是微秒：

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

通常我们并不关心时区信息，第二个参数都是 NULL，Nginx 为此定义了宏简化调用：

```
// 位于 os/unix/nginx_time.h
#define ngx_gettimeofday(tp) (void) gettimeofday(tp, NULL);
```

14.2 socket 系统调用

进程间通信有很多手段，例如信号、管道、文件、共享内存，但它们都局限于本机，而网络间通信则可以让运行在不同主机上的进程互相交换数据，跨越了空间的障碍。

网络间通信需要使用网络编程，而 socket 系列函数是事实上的网络编程标准，实现了目前应用最广泛的 TCP/IP 协议，同时也支持 UDP、UNIX Domain Socket 等协议。^①

socket 调用有阻塞 (blocking) 和非阻塞 (nonblocking) 两种模式，为了提高效率，在 Nginx 里使用的都是非阻塞模式，如果数据未准备好函数不会阻塞等待，而是立即返回 -1，同时 errno 置为 EAGAIN，这样进程就不会把时间浪费在等待上，可以处理其他的事情，最大效率地利用 CPU。

^① socket 一般译为“套接字”，但本书不使用这个中文术语，仍然使用英文原名。

本节只介绍 TCP 相关的函数，UDP 等其他协议暂不涉及。

14.2.1 socket

函数 `socket()` 创建一个 `socket`，返回表示 `socket` 的文件描述符：

```
int socket(int domain, int type, int protocol);    //创建 socket
```

`domain` 参数指定 `socket` 使用的协议族，常用的是 `AF_INET`、`AF_INET6` 和 `AF_UNIX`。

`type` 参数指定协议的类型，TCP 协议使用 `SOCK_STREAM`，UDP 协议使用 `SOCK_DGRAM`。

Nginx 还用宏重定义了 `socket` 描述符和函数：

```
// 位于 os/unix/nginx_socket.h
typedef int          ngx_socket_t;           //socket 描述符类型
#define ngx_socket  socket                   //创建 socket
```

14.2.2 bind

函数 `bind()` 把一个 `socket` 与具体的网络地址绑定：

```
int bind(int sockfd,                          //绑定 socket 使用的地址
         const struct sockaddr *addr, socklen_t addrlen);
```

`bind()` 通常用在服务器端，把 `socket` 绑定在公开的地址以对外提供服务。

14.2.3 listen

服务器端在绑定 `socket` 之后就可以调用 `listen()` 函数来监听客户端的连接：

```
int listen(int sockfd, int backlog);           //在指定的 socket 上监听连接
```

参数 `backlog` 指定系统内核里等待连接队列的最大长度。

14.2.4 accept

函数 `accept()` 从系统内核的监听队列里获得一个已经成功建立连接的 `socket`，客户端的地址则在 `addr` 和 `addrlen` 里给出：

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Linux 系统还提供一个特别的函数 `accept4()`，它比 `accept()` 多一个 `flags` 参数，如果指定为 `SOCK_NONBLOCK`，那么就直接返回一个已经是非阻塞的 `socket`，可以节省一次 `ioctl` 系统调用：

```
int accept4(int sockfd, struct sockaddr *addr, socklen_t *addrlen, int flags);
```

14.2.5 connect

`listen()`/`accept()` 函数用在服务器端被动接受连接，而 `connect()` 函数则用在客户端主动发起连接：

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

`connect()` 的接口参数与 `accept()` 非常相似，如果与服务器端三次握手成功那么 `sockfd` 就标识了此连接的 socket。

如果 `connect()` 是非阻塞的，那么函数调用后会立即返回，不会阻塞在连接握手过程中，同时 `errno` 被设置为 `EINPROGRESS`，表示已经开始连接但还未成功建立连接，需要之后再用 `getsockopt()` 确认。

14.2.6 recv

socket 本质上也是 UNIX 文件，所以可以用 `read()`/`write()` 读写数据，但专用的接口函数可以有更多的控制能力。

函数 `recv()` 从 socket 读取数据（实际上是内核里的缓冲区），也就是接收：

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

`recv()` 的返回值是读取到的数据长度，有以下三种情况：

- `>0` : 成功读取了一些数据到 `buf` 里，但不一定是 `len` 长度，可能要少；
- `=0` : 客户端已经关闭了连接；
- `-1` : 读取失败，最常见的 `errno` 是 `EAGAIN`，即数据还未准备好（没收到）。

14.2.7 send

函数 `send()` 向 socket 写入数据（实际上也是内核里的缓冲区），也就是发送。

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

`send()` 的返回值是成功发送的数据大小，如果小于长度 `len` 就表示此次没有完全发送数据（内核发送缓冲区已满），必须再执行下一次调用才可能发送完毕。

函数返回 `-1` 则表示发送失败，需要由 `errno` 确定具体原因，最常见的同样是 `EAGAIN`，这时只要简单地再次尝试就可能发送成功。

14.2.8 setsockopt

函数 `setsockopt()` 设置 socket 的各种属性, 适当调整选项可以优化网络连接:

```
int setsockopt(int sockfd, int level, int optname,  
               const void *optval, socklen_t optlen);
```

参数 `optname` 指定了要设置的属性, 常用的有:

- `SO_RCVBUF` : 设置 socket 的接收缓冲区大小;
- `SO_SNDBUF` : 设置 socket 的发送缓冲区大小;
- `SO_KEEPALIVE` : 保持 TCP 连接;
- `SO_REUSEADDR` : 允许重用 socket 地址, 解决 `TIME_WAIT` 问题;
- `SO_REUSEPORT` : 允许重用 socket 端口 (Linux 3.9 以上);
- `TCP_FASTOPEN` : TCP 协议专用, 启用 Fast Open 特性;
- `TCP_DEFER_ACCEPT` : TCP 协议专用, 启用 Deferred Accept 特性。

14.2.9 close

关闭 socket 连接需要使用 `close()` 函数, 它其实就是标准的关闭文件系统调用:

```
int close(int fd); //关闭 socket 连接
```

Nginx 用宏重定义了更清晰的名字:

```
// 位于 os/unix/nginx_socket.h  
#define ngx_close_socket close //关闭 socket 连接
```

14.2.10 函数关系图

socket 相关的函数较多, 但它们并不是完全彼此独立的, 而是有着先后的调用次序, 图 14-1 表示了它们的执行顺序。

注意本图描述的是非阻塞 socket 调用的顺序图, 与阻塞模式的 socket 调用略有不同。

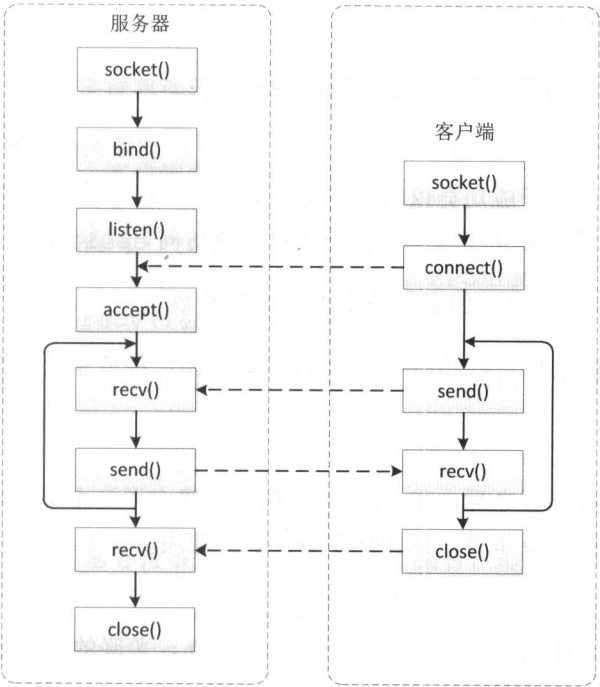


图 14-1 socket 相关函数的执行顺序

14.3 epoll 系统调用

现代操作系统都使用某种形式的 I/O 多路复用技术 (I/O Multiplexing) 来监控多个文件描述符处理 I/O，标准的 UNIX 系统提供 select 和 poll，但它们不能够应付高并发的场景，所以每个 UNIX 变体都有自己的高效系统调用：HPUX/Solaris 使用的是 eventport 或/dev/poll，FreeBSD 使用的是 kqueue，而 Linux 使用的就是 epoll。^①

epoll 优于 select 和 poll 的原因是它不需要维护文件描述符列表，也不用轮询事件，这些工作都交给操作系统在内核里完成，从而大大提高了效率，时间复杂度从 $O(n)$ 降低到了 $O(1)$ ，而且可监控的文件描述符数量几乎没有限制，能够轻易地支持百万级别的连接。^②

epoll 真正监控的是抽象的“文件描述符”——不仅是网络连接，也可以是管道或者其他类型的 UNIX 文件，但绝大多数情况下它们都是表示网络连接的 socket，出于简化叙述、明确含义的目的，本书之后会改用“socket”来代替“文件描述符”一词，请读者留意。

① 实际上 Linux 直到 2.6 内核才正式引入 epoll。
② 这里仅是对 epoll 工作原理的一个粗略描述，读者可参考其他资料了解详细的内部实现机制。

14.3.1 epoll_create

使用 `epoll` 首先要创建一个 `epoll` 的文件描述符:

```
int epoll_create(int size); //创建 epoll 对象, 返回描述符
```

`epoll_create()` 执行成功会返回一个文件描述符, 标志在内核里创建的 `epoll` 实例, 之后的 `epoll` 系列函数都要使用这个文件描述符才能访问 `epoll` 相关功能。

参数 `size` 目前没有意义, 但必须是个大于 0 的整数。

`epoll` 文件描述符同样可以使用 `close()` 函数关闭, 释放系统资源。

14.3.2 epoll_ctl

`epoll_create()` 执行之后, 在 Linux 内核里就建立了一个事件表, 用来记录需要监控的 `socket` 以及关联的 I/O 事件, 操作这个事件表要使用 `epoll_ctl()` 函数, 声明是:

```
int epoll_ctl(int epfd, int op, int sockfd, struct epoll_event *event);
```

函数向 `epfd` 表示的 `epoll` 实例添加一个 `socket`, 关联的事件则由参数 `op` 和 `event` 决定。

参数 `op` 是操作的类型, 有添加、修改和删除三种:

- `EPOLL_CTL_ADD` : 添加 `sockfd` 上关联的事件;
- `EPOLL_CTL_MOD` : 修改 `sockfd` 上关联的事件;
- `EPOLL_CTL_DEL` : 删除 `sockfd` 上关联的事件。

参数 `event` 的类型定义是:

```
typedef union epoll_data {
    void*                ptr; //epoll 事件的相关数据, 联合体
    ...                  //Nginx 只使用此成员
} epoll_data_t;           //其他成员有 fd、u32、u64
//epoll 事件的相关数据

struct epoll_event {
    uint32_t             events; //epoll 事件结构体
    epoll_data_t          data;  //事件标志位
    //事件相关数据
};
```

`epoll_event.data` 用来存储事件相关的数据, 当事件发生时用户可以从这里拿来直接使用, Nginx 只使用了 `ptr`, 保存连接对象的指针, 能够比其他字段存储更多的信息。

`epoll_event.events` 是一个标志量, 以位操作 (bit set) 指定了与 `socket` 关联

的具体事件，常用的有：^①

- EPOLLIN : 读事件，即 read ready;
- EPOLLOUT : 写事件，即 write ready;
- EPOLLRDHUP : 客户端关闭连接（断连）;

读事件是指 socket 有数据可读，此时如果调用 `recv()` 就可以接收到一些数据。写事件是指 socket 可写，调用 `send()` 就可以发送一些数据。EPOLLRDHUP 是一种特殊的读事件，如果客户端关闭连接，那么 `recv()` 会读取 0 个字节。

很显然，对于通常的 socket 连接应该全部关联这三个事件，这样一旦连接上可读或可写就会立即得到 epoll 的通知。

14.3.3 epoll_wait

函数 `epoll_wait()` 阻塞等待事件发生，返回事件集合，即获取内核的事件通知：

```
int epoll_wait(int epfd, struct epoll_event *events, //获取事件列表
               int maxevents, int timeout);
```

参数 `events` 是一个数组，长度为 `maxevents`，也就是说此次调用最多可以收集到 `maxevents` 个已经就绪（ready）的读写事件，实际的事件数量由函数的返回值决定，然后我们就可以遍历数组，对这些可读写的 socket 执行 `recv()` 或 `send()` 收发数据了。

参数 `timeout` 指示 `epoll_wait()` 阻塞等待的时间，单位是毫秒，但只有在没有任何事件发生时才会阻塞，如果内核事件表里有事件就不会阻塞而是立即返回，所以 `timeout` 只是个可能会阻塞等待的最长时间，而不是绝对的阻塞时间。

epoll 的高效体现在 Linux 内核只会拷贝 ready 事件，而通常情况下系统里只会有少量的活跃连接，大多数连接可能都阻塞在读写上，所以拷贝的数量很少。

14.3.4 LT 和 ET

关于 epoll 最著名的特性就是它的两种工作模式：LT 和 ET，LT 即 Level Triggered（水平触发），ET 即 Edge Triggered（边缘触发）。^②

LT 是 epoll 默认的工作模式，也被称为低速模式。在这种模式下，如果检测到的事件

① Linux 4.5 增加了 EPOLLEXCLUSIVE，在操作系统内核级别解决了“惊群”问题。

② 题外话，LT 和 ET 也是 Linux Server 开发常见的面试题之一。

ready 但未被处理会一直被触发（通过 `epoll_wait` 获取）。

ET 是高速模式，需要在 `epoll_ctl()` 函数添加事件时使用标志位“**EPOLLET**”启用，而且只支持非阻塞 socket 调用。如果事件 ready，内核只为此事件通知一次，应用程序必须立即处理事件对应的 socket 读写，但不管是否处理了事件，下一次 `epoll_wait()` 不会再次通知。相对于 LT 来说，ET 减少了事件被重复触发的次数，因而提高了效率。

我们使用一个简单的例子来说明这两种工作模式的区别。

假设现在有一个 socket，内核已经收到了发过来的 100 字节的数据，此时调用 `epoll_wait()` 就会得到事件通知，事件的类型是 EPOLLIN，即 read ready。

LT 模式下，调用 `recv()` 只读取 50 个字节，因为数据没有接收完，socket 仍然是可读的，所以再调用 `epoll_wait()` 还是会得到 EPOLLIN 事件通知，直至数据全部读取完毕。

ET 模式下，调用 `recv()` 同样只读取 50 个字节，虽然数据没有接收完，但再次调用 `epoll_wait()` 则不会得到通知，除非 socket 上又有新数据到来使事件变成 read ready。

所以 ET 模式下标准的操作方式被称为“贪婪”（greedy）：

- 读：只要可读，就一直读，直到返回 0（断连）或者 EAGAIN（内核接收缓冲区空）；
- 写：只要可写，就一直写，直到数据发送完毕或者 EAGAIN（内核发送缓冲区满）。

因为必须保证不能丢失客户端的连接，Nginx 仅对监听事件使用 LT 模式，其余的操作都使用 ET 模式，也就是使用 EPOLLET 标志位。

14.3.5 函数关系图

`epoll` 系列函数的调用关系可以用图 14-2 来表示。

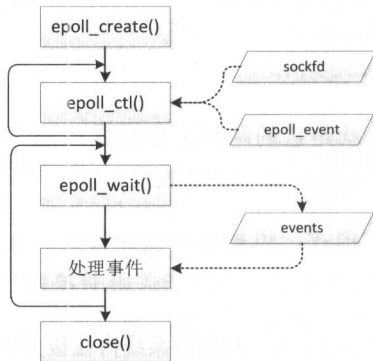


图 14-2 `epoll` 系列函数的调用关系

14.4 结构定义

Nginx 的事件机制是一个庞大的体系，不完全是由 event 模块来决定的，所以在介绍 event 模块之前，我们有必要了解一下 Nginx 框架提供的一些结构体，它们是事件机制得以运行的基础。

14.4.1 ngx_event_t

Nginx 使用结构体 ngx_event_t 屏蔽了不同操作系统的事件定义，可以简单地认为它代表了 Linux 里的结构体 epoll_event。下面仅列出与 Linux 相关的部分，省略了条件编译里 FreeBSD、Windows 等的代码：

```
// 位于 core/nginx_core.h
typedef struct ngx_event_s          ngx_event_t;    //简化类型定义

// 位于 event/nginx_event.h
struct ngx_event_s {
    void*          data;                //事件关联的对象，通常是 ngx_connection_t

    unsigned       write:1;             //读写事件的标志位，read or write
    unsigned       accept:1;            //监听事件的标志位，即 listen/accept
    unsigned       instance:1;          //检测事件是否失效的标志位
    unsigned       active:1;            //事件是否活跃，即被 epoll 监控
    unsigned       ready:1;             //事件已经就绪，也就是说有数据可读写
    unsigned       complete:1;          //异步操作的完成标志，用于 aio 和多线程
    unsigned       eof:1;               //当前的字节流已经结束，即客户端关闭连接
    unsigned       error:1;             //发生了错误
    unsigned       timedout:1;          //事件是否已经超时
    unsigned       timer_set:1;         //事件是否在定时器里
    unsigned       delayed:1;          //需要延迟处理，用于限速
    unsigned       deferred_accept:1;   //延后接收请求，提高运行效率
    unsigned       posted:1;           //事件是否已经加入延后处理队列，见 14.4.6 节
    unsigned       closed:1;           //事件已关闭
    unsigned       available:1;        //尽可能多地建立连接
    ngx_uint_t     index;              //通知机制里的简单的计数器

    ngx_event_handler_pt handler;       //重要！事件发生时调用的函数

    ngx_rbtree_node_t timer;           //红黑树节点成员，用于把事件加入定时器
    ngx_queue_t     queue;            //队列成员，加入延后处理的队列
};
```

ngx_event_t 使用位域的方式定义了大量的标志位，记录事件的状态，例如是否就绪、

是否失效、是否断连、是否出错等，比 `epoll` 简单的 `EPOLLIN/EPOLLOUT` 更加详细。

成员 `data` 的类型是 `void*` 指针，作用与 `epoll` 事件结构体 `epoll_event.data.ptr` 有些相似，用于保存事件相关的信息，通常是 Nginx 里的连接对象 `ngx_connection_t`。

成员 `handler` 是 `ngx_event_t` 里非常重要的字段，表示事件发生时的回调函数，是整个 Nginx 事件机制运行的关键，只有通过它才能驱动整个 Nginx 事件模型，它的定义是：

```
// 位于 core/nginx_core.h
typedef void (*ngx_event_handler_pt)(ngx_event_t *ev);
```

当事件发生时，Nginx 就会使用 `handler` 处理事件，也就是 `ev->handler(ev)`。

由于不可能无限地等待事件发生，必须要有超时机制，所以 Nginx 使用成员 `timer` 把 `ngx_event_t` 加入定时器红黑树，`timedout` 和 `timer_set` 标记超时的状态。

`ngx_event_t` 里还有一个重要标志位 `instance`，它用于检测当前事件是否失效。

14.4.2 ngx_connection_t

Nginx 使用结构体 `ngx_connection_t` 表示一个 TCP/UDP 的 `socket` 连接，定义摘要如下：

```
// 位于 core/nginx_core.h
typedef struct ngx_connection_s      ngx_connection_t; //简化类型定义

// 位于 core/nginx_connection.h
struct ngx_connection_s {
    void*                data;           //保存对应的 ngx_http_request_t 对象
    ngx_event_t*         read;           //对应的读事件
    ngx_event_t*         write;          //对应的写事件
    ngx_socket_t         fd;             //socket 描述符

    ngx_recv_pt          recv;           //接收数据的函数指针，通常是 ngx_recv
    ngx_send_pt          send;           //发送数据的函数指针，通常是 ngx_send
    ngx_recv_chain_pt    recv_chain;     //接收数据到多个缓冲区
    ngx_send_chain_pt    send_chain;     //发送数据从多个缓冲区

    ngx_listening_t*     listening;      //对应的监听对象

    off_t                sent;           //已经发送的字节数
    ngx_log_t*           log;            //用于记录日志的 log 对象
    ngx_pool_t*          pool;           //使用的内存池，默认大小是 256 字节
    int                  type;           //socket 类型，SOCK_STREAM 表示 TCP
```

```

struct sockaddr*   sockaddr;           //客户端的地址
ngx_str_t          addr_text;          //客户端地址的文本形式
struct sockaddr*   local_sockaddr;     //本地监听端口的地址

ngx_buf_t*         buffer;             //接收客户端数据的缓冲区
ngx_queue_t        queue;              //侵入式队列，用来重用连接对象
ngx_atomic_uint_t  number;             //连接的计数器
ngx_uint_t         requests;           //长连接上处理的请求次数，仅 HTTP 使用

unsigned           buffered:8;          //标志位，表示有数据缓冲待发送，即发送未完成
unsigned           timeout:1;           //是否已经超时
unsigned           error:1;            //是否已经出错
unsigned           destroyed:1;        //是否已经被销毁

unsigned           idle:1;             //连接处于空闲状态
unsigned           reusable:1;         //连接可以重用
unsigned           close:1;            //连接已经关闭，可以回收重用
};

```

`ngx_connection_t` 里的字段比较多，但都很重要，建议读者仔细阅读代码里的注释，了解这些字段的用途。

`ngx_connection_t` 代理了 `socket` 描述符，也就是成员 `fd`，`socket` 的类型由 `type` 字段标记（`SOCK_STREAM` 表示 TCP，`SOCK_DGRAM` 表示 UDP），当有读写事件发生时可以用成员 `read` 和 `write` 检查具体的状态，在它上面的读写操作则使用函数指针 `recv` 和 `send`，客户端的地址是 `sockaddr`，收到的数据保存在 `buffer` 里，发送的字节数记录在 `sent` 里。

`ngx_connection_t` 专门有一个内存池成员 `pool`，这样本次连接的所有内存会都在这个池里分配，当关闭连接时就可以直接销毁内存池释放内存，简化了内存的使用。

为了加快连接对象的分配使用，Nginx 把空闲的连接对象串成了一个单向链表，成员 `data` 就是链表的后继指针，但当连接对象已经被使用的时候 `data` 成员也没有被浪费，而是用于存储连接对应的 `ngx_http_request_t` 对象（在 Stream 机制里则是 `ngx_stream_session_t`），实现了 TCP 到 HTTP 的关联。

14.4.3 ngx_listening_t

Nginx 使用结构体 `ngx_listening_t` 代表 `socket` 系统调用 `listen()` 和 `accept()`，代码摘要如下：

```

// 位于 core/nginx_connection.h
typedef struct ngx_listening_s  ngx_listening_t;    //简化类型定义

```

```

struct ngx_listening_s {
    ngx_socket_t      fd;                // 监听用的 socket 描述符

    struct sockaddr*   sockaddr;         // 监听用的地址
    ngx_str_t          addr_text;        // 监听用的地址文本形式

    int                type;              // socket 的类型, SOCK_STREAM 表示 TCP
    int                backlog;           // backlog 队列, 即等待连接的队列
    int                rcvbuf;            // 内核接收缓冲区大小
    int                sndbuf;            // 内核发送缓冲区大小

    ngx_connection_handler_pt handler;    // 重要函数, accept 成功时的回调函数

    ngx_listening_t*   previous;          // 链表指针, 组成一个单向链表
    ngx_connection_t*  connection;        // 监听端口对应的连接对象
    ngx_uint_t         worker;            // 监听端口对应的 worker 进程序号

    unsigned            open:1;           // 监听端口已打开
    unsigned            remain:1;         // 重启时保持监听端口, 不关闭
    unsigned            bound:1;          // 因为总是绑定, 所以此标志位无意义
    unsigned            nonblocking_accept:1; // 暂未使用
    unsigned            listen:1;         // 监听端口正在监听
    unsigned            nonblocking:1;    // 暂未使用
    unsigned            ipv6only:1;       // 是否启用 IPv6
    unsigned            reuseport:1;      // 是否启用 reuseport 特性
    unsigned            deferred_accept:1; // 是否启用 Deferred Accept 特性
    int                fastopen;          // 是否启用 Fast Open 特性
};

```

可以看到, `ngx_listening_t` 里的字段基本上都是与监听端口有关的, Nginx 使用这些参数来决定监听时的 `backlog` 和 `reuseport`、`Deferred Accept`、`Fast Open` 等特性。

成员 `handler` 是成功 `accept` 连接时的回调函数, 它的声明是:

```

// 位于 core/nginx_core.h
typedef void (*ngx_connection_handler_pt)(ngx_connection_t *c);

```

监听端口关联的只有读事件, 也就是 `accept` 连接, 但 Nginx 也专门用一个连接对象来表示 (写事件被忽略), 当成功接受连接时就会触发 `connection->read->handler`, 它被设置为函数 `ngx_event_accept()`, 转而调用 `ngx_listening_t` 的 `handler`。

对于 `http` 模块, `handler` 指向 `ngx_http_init_connection()`, 即开始初始化 HTTP 连接, 启动 HTTP 处理机制。

14.4.4 ngx_cycle_t

ngx_cycle_t 作为 Nginx 框架的核心数据结构，保存了用来提供 Web 服务的监听端口数组和连接池，是 Nginx 处理网络连接的根本：

```
// 位于 core/nginx_cycle.h
struct ngx_cycle_s {
    ngx_array_t      listening;           // 监听端口数组

    ngx_connection_t* free_connections;    // 空闲连接链表，用于快速分配连接对象
    ngx_uint_t        free_connection_n;   // 空闲连接的数量
    ngx_queue_t        reusable_connections_queue; // 可重用的连接队列

    ngx_uint_t        connection_n;        // 连接池的大小，即总可用连接数
    ngx_connection_t* connections;         // 连接池，大小是 connection_n
    ngx_event_t*       read_events;         // 读事件池，与连接对应
    ngx_event_t*       write_events;        // 写事件池，与连接对应

    ...                                    // 其他成员
};
```

listening 是一个动态数组，里面存储了配置文件里所有用 listen 指令定义的监听端口对象 ngx_listening_t，Nginx 使用它来打开监听端口提供服务。listen() 和 bind() 调用都发生在 fork 多进程之前，所以每个 worker 进程都会监听相同的端口。

在高并发的情况下连接对象的反复创建与销毁是非常消耗资源的，所以 Nginx 采用了连接池技术——在程序启动时预先分配内存，创建出足够多的连接对象，之后就在这个连接池里重复利用连接，运行时使用对象几乎没有额外的成本。

连接池 connections 并不神秘，它只是一个简单的数组，大小 connection_n 由指令 “worker_connections” 确定，数组的创建是在启动时完成的，运行时不能改变，它静态限制了 Nginx（单个进程）的最大并发处理能力。

每一个连接对象都需要关联两个事件对象（读事件和写事件），所以连接池还需要两个配合工作的事件池：read_events 和 write_events，它们与 connections 是完全对应的，大小也是 connection_n。

连接池通常都很大，在重用时如果要遍历数组查找空闲连接对象效率会很低，所以 Nginx 使用 ngx_connection_t.data 把空闲连接串成了一个链表，成员 free_connections 指向链表的头节点，获取和归还连接对象只要操作头节点指针即可，巧妙地解决了对对象的分配回收问题，而且非常简单高效。

有两个函数用来操作连接池，获取和归还连接对象：

```
// 位于 core/nginx_connection.h
ngx_connection_t* ngx_get_connection(ngx_socket_t s, ngx_log_t *log);
void ngx_free_connection(ngx_connection_t *c);
```

Nginx 的连接池可以用图 14-3 来表示。

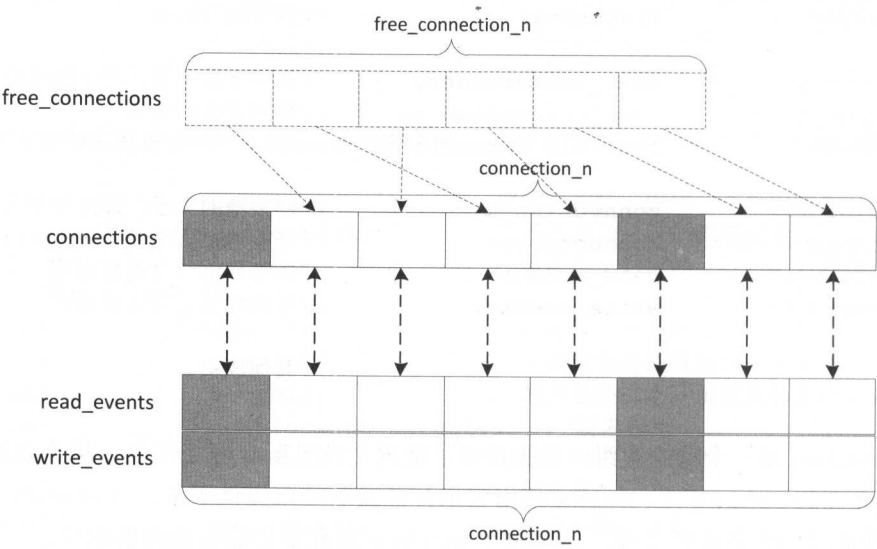


图 14-3 Nginx 的连接池

14.4.5 ngx_os_io_t

Nginx 使用结构体 ngx_os_io_t 封装了操作系统的 socket 数据收发功能，定义如下：

```
// 位于 os/unix/nginx_os.h
typedef struct {
    ngx_recv_pt      recv;           //TCP 接收数据
    ngx_recv_chain_pt recv_chain;    //TCP 接收多块数据
    ngx_recv_pt      udp_recv;      //UDP 接收数据
    ngx_send_pt      send;          //TCP 发送数据
    ngx_send_pt      udp_send;      //UDP 发送数据
    ngx_send_chain_pt udp_send_chain; //UDP 发送多块数据
    ngx_send_chain_pt send_chain;    //TCP 发送多块数据
    ngx_uint_t       flags;         //标志量，通常是 0
} ngx_os_io_t;
```

ngx_os_io_t 集合了 7 个 TCP/UDP 的数据收发函数，使用函数指针而不是固定函数的原因是可以针对不同的操作系统使用专门优化的实现来提高效率，非常灵活，每个具体的 UNIX

实现都有自己的 `ngx_os_io_t` 实例（桥接模式）。^①

`ngx_connection_t` 里的 `recv/send` 等函数指针就对应 `ngx_os_io_t` 的相应成员。

全局访问点

全局变量 `ngx_os_io` 提供函数指针默认实现：

```
// 位于 os/unix/nginx_posix_init.c
ngx_os_io_t ngx_os_io = {
    ngx_unix_recv,           //UNIX 收发函数的默认实现
    ngx_readv_chain,        //TCP 接收数据
    ngx_udp_unix_recv,      //TCP 接收多块数据
    ngx_unix_send,          //UDP 接收数据
    ngx_udp_unix_send,      //TCP 发送数据
    ngx_udp_unix_sendmsg_chain, //UDP 发送数据
    ngx_writev_chain,       //UDP 发送多块数据
    0,                      //TCP 发送多块数据
    0,                      //标志量
};
```

Nginx 对 Linux 有特别的“优待”，在初始化时会把 `send_chain` 指针替换为 `ngx_linux_sendfile_chain`，它使用 Linux 特有的 `sendfile` 系统调用，在处理磁盘文件时更高效。

不过 Nginx 并不直接使用 `ngx_os_io`，而是再定义了一个全局变量 `ngx_io`，它是这些函数真正的全局访问点，由 `event` 模块初始化为 `ngx_os_io`（见 14.8.4 节）：

```
// 位于 core/nginx_connection.c
ngx_os_io_t ngx_io;           //全局变量，操作系统的数据收发接口

// 位于 event/modules/nginx_epoll_module.c
ngx_io = ngx_os_io;          //设置数据收发函数的全局访问点
```

Nginx 还使用宏进一步屏蔽了收发函数的底层实现细节：

```
#define ngx_recv             ngx_io.recv           //TCP 接收数据
#define ngx_recv_chain       ngx_io.recv_chain     //TCP 接收多块数据
#define ngx_udp_recv         ngx_io.udp_recv       //UDP 接收数据
#define ngx_send             ngx_io.send           //TCP 发送数据
#define ngx_send_chain       ngx_io.send_chain     //TCP 发送多块数据
#define ngx_udp_send         ngx_io.udp_send       //UDP 发送数据
#define ngx_udp_send_chain   ngx_io.udp_send_chain //UDP 发送多块数据
```

^① 如果愿意，我们完全可以实现自己的专用 `recv/send` 函数，并且在运行时动态替换。

收发函数的实现

`ngx_unix_recv()` 和 `ngx_unix_send()` 是 Nginx 最基本的数据收发函数，执行的前提是 `socket` 可读或可写（`EPOLLIN` 或 `EPOLLOUT`，`ev->ready==1`），代码很好地示范了非阻塞 `socket` 的用法，值得我们研究学习。

`ngx_unix_recv()` 返回 0 表示客户端断连，大于 0 是接收到数据，小于 0 则是出错，函数内部处理了 `EAGAIN`，会反复读取：^①

```
// 位于 os/unix/ngx_recv.c
ssize_t ngx_unix_recv(ngx_connection_t *c, u_char *buf, size_t size)
{
    rev = c->read; // 获取连接的事件

    do {
        n = recv(c->fd, buf, size, 0); // 执行系统调用 recv 读数据

        if (n == 0) { // 0 字节，客户端断连
            rev->ready = 0; // 读事件不可用
            rev->eof = 1; // eof，即连接关闭
            return 0; // 返回读取的字节数，0
        }

        if (n > 0) { // 读取了 n 字节的数据
            return n; // 返回读取的字节数 n
        }

        err = ngx_socket_errno; // n<0，检查错误码

        if (err == NGX_EAGAIN || err == NGX_EINTR) { // EAGAIN 和 EINTR 不算是错误
            n = NGX_AGAIN; // 返回 EAGAIN，可以重试读取
        } else { // 其他错误码都是错误，退出循环
            n = ngx_connection_error(c, err, "recv() failed");
            break;
        }
    } while (err == NGX_EINTR); // 如果被系统中断则退出循环

    rev->ready = 0; // 读事件不可用

    if (n == NGX_ERROR) { // 不可恢复的错误
        rev->error = 1; // 设置事件的错误标志位
    }
}
```

① 代码里省略了 `kqueue`、`EPOLLRDHUP` 等部分。

```

}
return n;                                //返回错误码
}

```

ngx_unix_send() 的逻辑比 ngx_unix_recv() 要简单一些, 只有两种情况, 大于 0 是发送成功, 其他的都是出错:

```

// 位于 os/unix/ngx_send.c
ssize_t ngx_unix_send(ngx_connection_t *c, u_char *buf, size_t size)
{
    wev = c->write;                        //获取连接的写事件

    for ( ;; ) {
        n = send(c->fd, buf, size, 0);      //执行系统调用 send 发送数据

        if (n > 0) {                        //成功发送了 n 个字节
            if (n < (ssize_t) size) {      //没有发送完所有数据
                wev->ready = 0;            //暂时不可写 (内核缓冲区已满)
            }
            c->sent += n;                    //记录连接上发送的总字节数
            return n;                       //返回本次发送的字节数 n
        }

        err = ngx_socket_errno;            //n<=0, 检查错误码

        if (n == 0) {                       //n=0, 数据没发出去
            wev->ready = 0;                 //暂时不可写 (内核缓冲区已满)
            return n;                       //返回本次发送的字节数 0
        }

        if (err == NGX_EAGAIN || err == NGX_EINTR) { //EAGAIN 和 EINTR 不算错误
            wev->ready = 0;                 //暂时不可写
            if (err == NGX_EAGAIN) {        //如果错误是 EAGAIN 可以再试一次
                return NGX_AGAIN;          //返回 EAGAIN 要求重试
            }
        } else {                            //其他错误码都是错误, 退出循环
            wev->error = 1;                 //设置事件的错误标志位
            return NGX_ERROR;              //返回错误码
        }
    }
}

```

14.4.6 ngx_event_actions_t

与 ngx_os_io_t 类似, Nginx 使用结构体 ngx_event_actions_t 封装了操作系统

的 I/O 多路复用机制，也是个函数指针集合，定义如下：

```
// 位于 event/nginx_event.h
typedef struct {
    ngx_int_t  (*add) (ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
    ngx_int_t  (*del) (ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

    ngx_int_t  (*enable) (ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
    ngx_int_t  (*disable) (ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

    ngx_int_t  (*add_conn) (ngx_connection_t *c);
    ngx_int_t  (*del_conn) (ngx_connection_t *c, ngx_uint_t flags);

    ngx_int_t  (*notify) (ngx_event_handler_pt handler);

    ngx_int_t  (*process_events) (ngx_cycle_t *cycle, ngx_msec_t timer,
                                ngx_uint_t flags);

    ngx_int_t  (*init) (ngx_cycle_t *cycle, ngx_msec_t timer);
    void        (*done) (ngx_cycle_t *cycle);
} ngx_event_actions_t;           //事件机制的访问接口，是一个函数表
```

初看上去结构体似乎很复杂，这是因为它没有使用 typedef 声明函数指针原型，而是直接在结构体内部定义了函数指针。

我们可以用 typedef 定义函数指针类型，改写为较容易理解的形式：

```
typedef ngx_int_t  (*events_ctl_pt) (...);           //修改事件
typedef ngx_int_t  (*events_add_all_pt) (...);       //添加连接上的读写事件
typedef ngx_int_t  (*events_del_all_pt) (...);       //删除连接上的读写事件
typedef ngx_int_t  (*events_notify_pt) (...);        //事件通知
typedef ngx_int_t  (*events_process_pt) (...);       //收集并处理事件
typedef ngx_int_t  (*events_init_pt) (...);          //初始化事件机制
typedef void        (*events_done_pt) (...);         //结束事件机制的收尾工作

typedef struct {
    events_ctl_pt      add;           //添加一个事件
    events_ctl_pt      del;           //删除一个事件
    events_ctl_pt      enable;        //启用一个事件
    events_ctl_pt      disable;       //禁用一个事件
    events_add_all_pt  add_conn;      //添加连接上的读写事件
    events_del_all_pt  del_conn;      //删除连接上的读写事件
    events_notify_pt   notify;        //事件通知
    events_process_pt  process_events; //收集并处理事件
    events_init_pt     init;          //初始化事件机制
```

```
events_done_pt      done;                //结束事件机制的收尾工作
} ngx_event_actions_t;
```

现在就可以很明显地看出 `ngx_event_actions_t` 里有 10 个函数指针，都是用来操作事件的，具体作用是（以 `epoll` 为例）：

- `add` : 在 `epoll` 里增加一个事件，之后调用 `epoll_wait` 就可获得通知；
- `del` : 在 `epoll` 里删除一个事件，之后调用 `epoll_wait` 不能获得通知；
- `enable` : 目前同 `add`；
- `disable` : 目前同 `del`；
- `add_conn` : 向 `epoll` 增加连接上的读写事件，相当于两次 `add`；
- `del_conn` : 从 `epoll` 删除连接上的读写事件，相当于两次 `del`；
- `notify` : 事件通知机制，使用 `eventfd`，目前仅多线程使用，见第 15 章；
- `process_events`: 收集并处理之前添加的事件，相当于 `epoll_wait`；
- `init` : 初始化事件机制，相当于 `epoll_create`；
- `done` : 结束事件机制的收尾工作，相当于 `close`。

`ngx_event_actions_t` 屏蔽了 `select/poll/epoll/kqueue` 等各种事件模型之间的差异，Nginx 用这些函数就可以一致地增加、修改、删除和处理事件。

Nginx 使用一个全局变量 `ngx_event_actions` 作为事件处理函数的入口：

```
// 位于 event/nginx_event.c
ngx_event_actions_t  ngx_event_actions;    //事件处理函数的入口
```

与 `ngx_recv/ngx_send` 类似，Nginx 定义了几个宏来简化调用：

```
// 位于 event/nginx_event.h
#define ngx_process_events  ngx_event_actions.process_events
#define ngx_done_events    ngx_event_actions.done

#define ngx_add_event      ngx_event_actions.add
#define ngx_del_event      ngx_event_actions.del
#define ngx_add_conn       ngx_event_actions.add_conn
#define ngx_del_conn       ngx_event_actions.del_conn

#define ngx_notify         ngx_event_actions.notify
```

14.4.7 ngx_posted_events

Nginx 每调用一次 `ngx_process_events` 都会收集到大量的事件，因为 Nginx 是单线程的，所有的事件都必须顺序逐个处理，为了加快事件的处理速度，避免处理事件时间过长造

成的等待, Nginx 定义了两个队列, 用来存放暂时不需要“立即”处理的事件:

```
// 位于 event/nginx_event_posted.c
ngx_queue_t  ngx_posted_accept_events; //accept 事件, 即客户端发起的连接请求
ngx_queue_t  ngx_posted_events;       //其他事件, 即读写事件和通知事件
```

使用两个队列分别存放事件的原因是通常客户端的 accept 连接事件更紧急, 处理的优先级更高, 这实际上是一种简化的优先队列设计, 可以让 Nginx 及时响应客户端的连接请求。^①

ngx_event_t 本身已经具备了加入延后处理队列的条件, queue 字段用来把事件加入队列, 标志位 posted 标记事件是否已经加入, 代码再摘录如下:

```
// 位于 event/nginx_event.h
struct ngx_event_s {
    unsigned        posted:1;           //事件是否已经加入延后处理队列
    ngx_queue_t     queue;              //队列成员, 用于加入延后处理队列
    ...                               //其他成员
};
```

Nginx 定义了两个函数宏来简化队列操作:

```
// 位于 event/nginx_event_posted.h
#define ngx_post_event(ev, q)          //把事件加入延后处理队列
#define ngx_delete_posted_event(ev)    //把事件从延后处理队列里移除
```

函数 ngx_event_process_posted() 遍历队列, 调用 handler 逐个处理事件:

```
// 位于 event/nginx_event_posted.c
void
ngx_event_process_posted(ngx_cycle_t *cycle, ngx_queue_t *posted)
{
    while (!ngx_queue_empty(posted)) { //遍历队列, 直至队列为空

        q = ngx_queue_head(posted);    //取队列头节点
        ev = ngx_queue_data(q, ngx_event_t, queue); //偏移量计算得到事件
        ngx_delete_posted_event(ev);    //即将处理事件, 移出队列

        ev->handler(ev);                //调用 handler 处理事件
    }
}
```

^① 所以 ngx_posted_accept_events 里也可以存放其他事件, 会优先得到处理的机会。

14.4.8 结构关系图

上面我们介绍了七个事件机制相关的数据结构，这些数据结构之间也存在着密切的联系，彼此用指针互相关联，如图 14-4 所示。

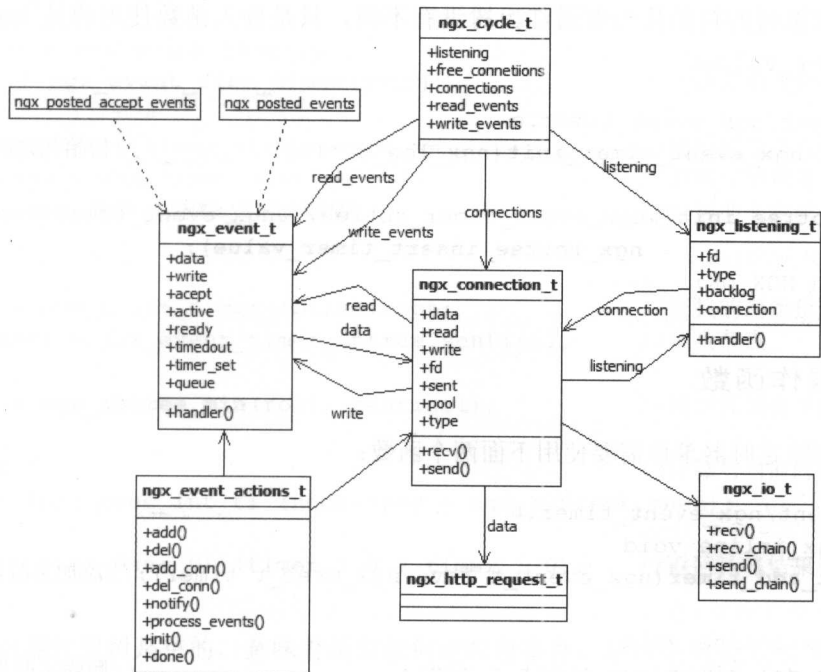


图 14-4 事件机制数据结构的关系

14.5 定时器

定时器是事件驱动模型里的重要概念，用于检测超时事件（timedout，也称为过期 expire），我们也可以灵活使用定时器延后执行某些费时的操作，提高系统的整体响应效率。

定时器不属于 I/O 事件，所以不能用 epoll、kqueue 等系统调用来处理。常用的定时器实现方式有时间链表、时间轮等，而 Nginx 则利用了时间是有序的这一特点，使用红黑树结构管理所有定时器对象，效率很高，可以精确到毫秒级别。

14.5.1 红黑树

由于 ngx_event_t 结构体里已经有了红黑树节点成员 timer（14.4.1 节），所以使用时间戳（单位是毫秒）作为 key 就可以把需要超时检测的事件对象组织成一棵红黑树。

Nginx 使用全局变量 `ngx_event_timer_rbtrees` 表示定时器红黑树:

```
// 位于 event/nginx_event_timer.c
ngx_rbtrees_t          ngx_event_timer_rbtrees;    //定时器红黑树
static ngx_rbtrees_node_t  ngx_event_timer_sentinel; //红黑树哨兵节点
```

定时器红黑树的初始化与普通红黑树没有不同, 只是插入函数使用的是 `ngx_rbtrees_insert_timer_value`:

```
// 位于 event/nginx_event_timer.c
ngx_int_t ngx_event_timer_init(ngx_log_t *log)      //初始化定时器红黑树
{
    ngx_rbtrees_init(&ngx_event_timer_rbtrees, &ngx_event_timer_sentinel,
                     ngx_rbtrees_insert_timer_value);
    return NGX_OK;
}
```

14.5.2 操作函数

添加或删除定时器事件需要使用下面两个函数:

```
// 位于 event/nginx_event_timer.h
static ngx_inline void
ngx_event_add_timer(ngx_event_t *ev, ngx_msec_t timer); //添加定时器, 毫秒单位

static ngx_inline void
ngx_event_del_timer(ngx_event_t *ev);                  //删除定时器
```

这两个函数使用 `ngx_event_t.timer` 成员, 在定时器红黑树里添加或删除一个节点, 同时会设置 `timer_set` 成员, 标记事件是否已经在红黑树里, 即是否需要检测超时。

同样的, Nginx 也定义了两个宏来简化定时器操作:

```
// 位于 event/nginx_event.h
#define ngx_add_timer      ngx_event_add_timer      //添加定时器
#define ngx_del_timer      ngx_event_del_timer      //删除定时器
```

14.5.3 超时处理

socket 的读写不总是立即可用的, 很多情况下需要等待但又不能无限等待, 所以就必须把事件标记一个超时时间, 通常是若干毫秒, 然后加入定时器红黑树, 当时间到而事件还没有发生时就意味着超时。

处理超时事件可分为两步: 首先查找超时事件, 然后处理超时事件。

查找超时事件

在定时器红黑树里所有的事件按照超时的时间从小到大排序，所以只需要取出最小节点，用它的超时时间(即 key)与当前时间比较就可以知道是否存在超时事件，在 Nginx 里使用的函数是 `ngx_event_find_timer()`：

```
// 位于 event/nginx_event_timer.c
ngx_msec_t ngx_event_find_timer(void)           //查找是否有超时事件
{
    if (ngx_event_timer_rbtrees.root ==         //比较根节点与哨兵节点
        &ngx_event_timer_sentinel) {           //判断红黑树是否为空
        return NGX_TIMER_INFINITE;              //空树即无定时器，不超时
    }

    root = ngx_event_timer_rbtrees.root;         //查找的起始位置，根节点
    sentinel = ngx_event_timer_rbtrees.sentinel; //哨兵节点

    node = ngx_rbtrees_min(root, sentinel);      //找到红黑树里的最小节点

    timer =                                       //计算与当前时间的毫秒差值
        (ngx_msec_int_t) (node->key - ngx_current_msec);

    return (ngx_msec_t) (timer > 0 ? timer : 0); //返回计算结果
}
```

如果定时器红黑树是空的，意味着没有任何定时器事件，所以函数会返回 `NGX_TIMER_INFINITE`，也就是-1，表示没有超时事件。

如果红黑树最小节点的 key 大于当前时间，函数返回两者的差值就是一个正数 timer，表示当前时间距离最近一个事件的超时时间，也就是说还没有超时事件（不必着急处理），但下一个事件即将在 timer 毫秒后到期。Nginx 会使用它作为 `epoll_wait()` 的等待时间，即如果无任何连接读写时间发生，`epoll_wait()` 将阻塞等待最多 timer 毫秒。

如果红黑树最小节点的 key 小于当前时间，函数不会返回差值而直接给出 0。这种情况下意味着至少有一个事件已经超时过期了（因为预计的时间已经落后于当前时间），必须尽快在红黑树里找出所有的过期事件处理。

处理超时事件

函数 `ngx_event_expire_timers()` 遍历定时器红黑树，找出所有的过期事件，调用它里面的 handler 处理，与 `ngx_event_process_posted()` 类似：

```
void ngx_event_expire_timers(void)           //处理过期事件
```



```

{
    sentinel = ngx_event_timer_rbtrees.sentinel;           //哨兵节点

    for ( ;; ) {                                           //无限循环查找过期节点
        root = ngx_event_timer_rbtrees.root;              //红黑树根节点
        if (root == sentinel) { return; }                 //红黑树为空则循环结束

        node = ngx_rbtrees_min(root, sentinel);           //取最小节点, 可能是过期的
        if ((ngx_msec_int_t) (node->key - ngx_current_msec) > 0) {
            return;                                         //大于当前时间, 没有过期, 查找完毕, 退出
        }

        //offsetof 宏利用偏移量计算得到事件对象
        ev = (ngx_event_t *) ((char *) node - offsetof(ngx_event_t, timer));

        ngx_rbtrees_delete(&ngx_event_timer_rbtrees, &ev->timer); //从红黑树里删除
        ev->timer_set = 0;                                         //清除定时器标志位

        ev->timedout = 1;                                         //设置事件超时标志位
        ev->handler(ev);                                          //回调函数, 处理超时
    }
}

```

函数的逻辑并不难理解, 反复查找红黑树里的最小节点, 直至遇到不超时的节点为止, 对于所有超时的事件设置 `timedout` 标志位, 告诉事件已经超时过期, 然后调用 `handler` 执行事件自己的超时处理逻辑, 流程图如图 14-5 所示。

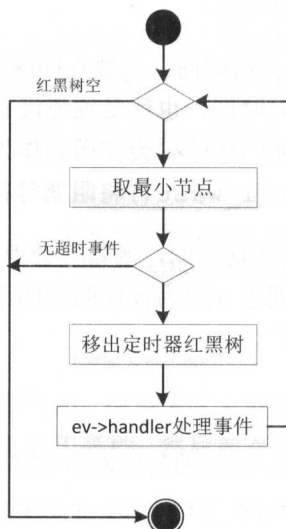


图 14-5 超时事件的处理逻辑

超时检查的成本很低，也很重要，所以 Nginx 的各种事件处理函数 handler 通常首先会检查是否超时，代码的基本形式是：

```
if (ev->timedout) {
    ev->timedout = 0;
    ...
}
```

//检查超时标志位
//已超时那么先清空
//具体的处理超时逻辑

14.6 模块体系

event 模块是 Nginx 框架里的一类重要模块，类型是 `NGX_EVENT_MODULE("EVNT")`，它与进程机制密切配合，为 http、stream 等模块提供运行的动力。

event 模块体系里的核心模块有两个：`ngx_events_module` 和 `ngx_event_core_module`，它们定义和管理了所有其他的 event 模块。

event 模块的主要目的是封装不同操作系统的事件处理机制，所以比起 http 模块来说数量相当少，当前仅有 7 个，分别代表了 `select/poll/epoll/kqueue/devpoll/eventport/win32_select` 这几种 I/O 多路复用技术，本书重点讲解 `ngx_epoll_module`。^①

与我们之前接触的 http 模块不同，event 模块是一类全新的 Nginx 模块，它的结构定义和组织管理差异较大，所以最好结合第 6 章阅读本节。

14.6.1 函数指针表

遵从 Nginx 的模块架构，event 模块也定义了自己的函数指针表 `ngx_event_module_t`，用于“子类化”模块结构体 `ngx_module_t` 里的 `ctx` 字段：

```
// 位于 event/nginx_event.h
typedef struct {
    ngx_str_t*      name;          //模块的名字，用于 use 指令
    void*           (*create_conf)(ngx_cycle_t *cycle);
    char*           (*init_conf)(ngx_cycle_t *cycle, void *conf);

    ngx_event_actions_t actions;    //事件处理机制的各种函数指针
} ngx_event_module_t;
```

^① 早期的 Nginx 还提供 `rtsig`、`aio` 两个 event 模块，但在 1.9.x 之后已不再支持。

`ngx_event_module_t` 的形式与 `ngx_http_module_t` 很相似,主要是一些函数指针,但 `ngx_http_module_t` 里的函数指针都是配置解析相关的,而 `ngx_event_module_t` 里只有两个配置解析相关函数指针,剩下的就是封装操作系统的事件处理机制的 `actions`。

因为 `event` 模块的配置比较简单,没有 `http` 那么复杂的层次,所以配置解析函数只需要两个,分别用来创建配置结构体和初始化配置结构体。

字段 `actions` 的类型就是我们在 14.4.6 节介绍的 `ngx_event_actions_t`,它里面有 10 个函数指针,抽象了对事件的添加、删除和处理等操作,每个具体的 `event` 模块都会实现这些函数,从而提供对操作系统 I/O 多路复用技术的完整封装。

14.6.2 模块的组织形式

`ngx_events_module` 属于 `core` 模块,组织所有的 `event` 模块,为它们在 `ngx_cycle->conf_ctx` 里创建存储配置结构体的内存空间。

`ngx_events_module` 只提供一个配置指令“`events`”,用来解析 `events{...}` 配置块,配置所有的 `event` 模块,函数 `ngx_events_block()` 代码摘要如下:

```
// 位于 event/nginx_event.c
void***          ctx;                //指向存储配置结构体数组的指针,注意有三个*

ngx_event_max_module =                //计算 event 模块的数量,初始化 ctx_index
    ngx_count_modules(cf->cycle, NGX_EVENT_MODULE);

ctx = ngx_palloc(cf->pool, sizeof(void *)); //void***指针,存储所有配置

*ctx = ngx_palloc(                //为所有 event 模块分配存储空间,是个指针数组
    cf->pool, ngx_event_max_module * sizeof(void *));

*(void **) conf = ctx;                //存放到 cycle->conf_ctx 里

for (i = 0; cf->cycle->modules[i]; i++) {    //遍历 cycle 里的模块数组
    if (cf->cycle->modules[i]->type != NGX_EVENT_MODULE) { //检查 event 模块
        continue;                                //不是 event 模块则跳过
    }

    m = cf->cycle->modules[i]->ctx;            //获取模块的 ctx 函数表
    if (m->create_conf) {                    //模块是否有 create 函数
        (*ctx)[cf->cycle->modules[i]->ctx_index] = //创建模块的配置数据
            m->create_conf(cf->cycle);
    }
}
```

```
pcf = *cf; //暂存之前配置解析的 ctx
cf->ctx = ctx; //设置本配置块的 ctx
cf->module_type = NGX_EVENT_MODULE; //设置模块的类型标志量
cf->cmd_type = NGX_EVENT_CONF; //设置命令的类型标志量

rv = ngx_conf_parse(cf, NULL); //使用新的 ctx 解析块内指令

*cf = pcf; //解析完毕，恢复之前配置解析的 ctx

for (i = 0; cf->cycle->modules[i]; i++) { //遍历 cycle 里的模块数组
    if (cf->cycle->modules[i]->type != NGX_EVENT_MODULE) { //检查 event 模块
        continue; //不是 event 模块则跳过
    }

    m = cf->cycle->modules[i]->ctx;
    if (m->init_conf) { //模块是否有 init 函数
        rv = m->init_conf(cf->cycle, //初始化模块的配置
            (*ctx)[cf->cycle->modules[i]->ctx_index]);
    }
}
```

event 模块的配置都很简单，没有层次关系，每个模块只用一个结构体就可以保存所有的配置信息，这些结构体以 void*指针的形式存储，所以就是一个指针数组，也就是 void**，而指向这个数组的指针 ctx 的类型自然就是“(void**) *”。

解析函数的入口参数 conf 是 cf->cycle->conf_ctx 为 ngx_events_module 提供的存储位置，它的类型是 void*，所以需要强制转型为 void**再存储 ctx。

创建了配置结构体数组之后，ngx_events_module 按部就班地调用每个 event 模块的 create_conf，递归解析指令进行模块各自的配置，最后再调用模块的 init_conf，完成所有 event 模块的配置初始化。

event 模块配置数据的存储模型如图 14-6 所示。

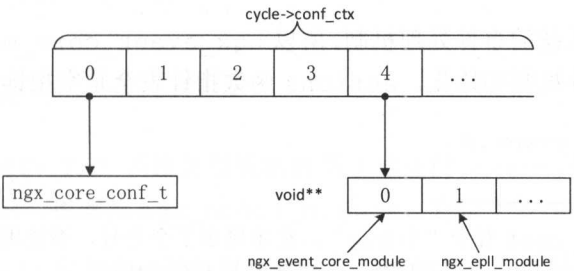


图 14-6 event 模块配置数据的存储模型

根据这个存储模型，我们就可以较容易地理解获取 event 模块配置的宏：^①

```
// 位于 core/nginx_conf_file.h
#define ngx_get_conf(conf_ctx, module) conf_ctx[module.index]

// 位于 event/nginx_event.h
#define ngx_event_get_conf(conf_ctx, module) \
    (*(ngx_get_conf(conf_ctx, ngx_events_module))) [module.ctx_index];
```

宏 ngx_event_get_conf 先使用 ngx_get_conf 得到数组里 ngx_events_module 所在位置的指针，类型为 void***，再使用解引用操作符 (*) 后是 void**，这就成为了一个存储 void* 的数组，再使用具体模块的 ctx_index 作为索引，就能够得到模块的配置结构体指针 void*。

14.6.3 核心配置

ngx_event_core_module 负责管理所有具体的 event 模块，搭建 event 整体框架，作用与 ngx_http_core_module 类似。

ngx_event_core_module 的配置结构体定义是：

```
// 位于 event/nginx_event.h
typedef struct {
    ngx_uint_t    connections; //进程可使用的连接数量，即 cycle 里的连接池大小
    ngx_uint_t    use;         //使用的 event 模块，即模块的 ctx_index
    ngx_flag_t    multi_accept; //是否尽可能多接受客户端请求，影响进程间负载均衡
    ngx_flag_t    accept_mutex; //是否使用进程间负载均衡锁
    ngx_msec_t    accept_mutex_delay; //负载均衡锁的等待时间
    u_char*       name;         //使用的 event 模块的名字
} ngx_event_conf_t;
```

配置结构体里的成员基本对应 events{} 配置块里的指令，是 Nginx 事件机制运行的核心参数，其中最重要的就是 connections，它直接决定了 Nginx 连接池的大小，也就是单个进程的最大并发处理能力。

由于不封装操作系统的事件处理机制，所以 ngx_event_core_module 的主要工作就是配置数据的创建和事件机制初始化，actions 函数指针表全是空指针：

```
// 位于 event/nginx_event.c
```

① 宏 ngx_event_get_conf 有个“小 bug”，在末尾多了个分号，不能用在表达式里。不过通常很少有人会使用这个宏去操作 event 模块，所以暂时没有什么影响。

```
static ngx_event_module_t  ngx_event_core_module_ctx = {
    &event_core_name,                //模块的名字是"event_core"
    ngx_event_core_create_conf,      //create configuration
    ngx_event_core_init_conf,        //init configuration

    { NULL, NULL, NULL, ... , NULL } //全空指针!
};
```

函数 `ngx_event_core_init_conf()` 用来决定配置数据的默认值，并把关键参数 `connections` 拷贝到 `cycle` 里供其他模块使用，代码摘要如下：

```
// 位于 event/nginx_event.c
static char *
ngx_event_core_init_conf(ngx_cycle_t *cycle, void *conf)
{
    ngx_event_conf_t*      ecf = conf;          //配置结构体指针
    ngx_module_t*          module;              //默认使用的模块指针

    module = &ngx_epoll_module;                //Linux 默认使用 epoll

    ngx_conf_init_uint_value(                    //默认连接池大小是 512
        ecf->connections, DEFAULT_CONNECTIONS);
    cycle->connection_n = ecf->connections;      //拷贝到 cycle 的 connection_n

    ngx_conf_init_uint_value(                    //设置 use 为模块的 ctx_index
        ecf->use, module->ctx_index);

    event_module = module->ctx;                  //设置使用的模块的名字
    ngx_conf_init_ptr_value(ecf->name, event_module->name->data);

    ngx_conf_init_value(ecf->multi_accept, 0);   //默认一次只 accept 一个连接
    ngx_conf_init_value(ecf->accept_mutex, 0);   //默认关闭负载均衡锁
    ngx_conf_init_msec_value(                    //默认负载均衡锁的等待时间是 500 毫秒
        ecf->accept_mutex_delay, 500);

    return NGX_CONF_OK;
}
```

14.6.4 epoll 模块

Nginx 为多种不同的 I/O 多路复用机制提供了相应的 event 模块实现，本书只关注 Linux 上的 `epoll` 机制，也就是 `ngx_epoll_module`，它位于 `event/modules` 目录。

`ngx_epoll_module` 也有自己的配置结构体，但通常我们无须特别配置：

```
// 位于 event/modules/nginx_epoll_module.c
typedef struct {
    ngx_uint_t      events;                //epoll_wait 的数组长度
    ngx_uint_t      aio_requests;          //aio 操作的最大数量
} ngx_epoll_conf_t;                       //epoll 模块的配置结构体
```

epoll 模块的核心是函数指针表，它基于 Linux epoll 系统调用实现了 ngx_event_actions_t 里的 10 个函数指针：

```
// 位于 event/modules/nginx_epoll_module.c
static ngx_event_module_t ngx_epoll_module_ctx = {
    &epoll_name,                          //"epoll"
    ngx_epoll_create_conf,                 //create configuration
    ngx_epoll_init_conf,                   //init configuration
    {
        ngx_epoll_add_event,               //add an event
        ngx_epoll_del_event,               //delete an event
        ngx_epoll_add_event,               //enable an event
        ngx_epoll_del_event,               //disable an event
        ngx_epoll_add_connection,          //add an connection
        ngx_epoll_del_connection,          //delete an connection
        ngx_epoll_notify,                  //trigger a notify
        ngx_epoll_process_events,           //process the events
        ngx_epoll_init,                    //init the events
        ngx_epoll_done,                     //done the events
    }
};
```

这些函数直接实现了 Nginx 的事件机制，也就是事件处理机制的入口 ngx_event_actions，详细的内部逻辑将在稍后的 14.8 节介绍。

epoll 模块的定义是：

```
ngx_module_t ngx_epoll_module = {
    NGX_MODULE_V1,
    &ngx_epoll_module_ctx,                /* module context */
    ngx_epoll_commands,                   /* module directives */
    NGX_EVENT_MODULE,                     /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
```

```
};
```

注意它的 7 个函数指针都是空的，但这并不意味着在 Nginx 启动时不会做任何事情，实际上会由 `ngx_event_core_module` 调用 `actions.init` 进行初始化。

14.7 全局变量

与进程机制类似，Nginx 事件机制也使用了一些全局变量，用来更新时间、标记事件机制和进程间负载均衡的状态，还有统计连接信息。

14.7.1 更新时间相关

随时都能够获取准确的时间是一个 Web 服务器最基本的功能。单纯地获取时间值很容易，只要调用 `gettimeofday()` 就可以，但为了获取准确的时间而频繁地执行系统调用就会降低效率，所以 Nginx 提出了“时间精确度”的概念，可以用信号 `SIGALRM` 来保证精确度：

```
// 位于 event/nginx_event.c
static ngx_uint_t      ngx_timer_resolution;      //时间精确度，单位是毫秒
sig_atomic_t          ngx_event_timer_alarm;      //收到信号 SIGALRM 的标志
```

`ngx_timer_resolution` 不由 `event` 模块决定，而是由核心配置结构体 `ngx_core_conf_t` 的 `timer_resolution` 决定（见 13.4.2 节）。^①

如果在配置文件里使用了指令“`timer_resolution`”，那么 Nginx 就会使用系统调用 `setitimer()`，定时发送 `SIGALRM`，信号的处理函数就会设置变量 `ngx_event_timer_alarm`：

```
// 位于 event/nginx_event.c
static void ngx_timer_signal_handler(int signo)    //SIGALRM 的处理函数
{
    ngx_event_timer_alarm = 1;                    //设置变量，系统闹钟标志
}
```

不过大多数情况下 Nginx 每次处理事件时都会更新时间，所以无须特别设置时间精确度，只有当服务比较空闲时它才有意义。

^① 这一点比较奇怪，也许以后 Nginx 会有所改变。

14.7.2 事件机制相关

Nginx 使用两个全局变量来记录当前使用的事件机制的状态：

```
// 位于 event/nginx_event.c
ngx_uint_t          ngx_event_flags;           //事件机制的状态
ngx_event_actions_t ngx_event_actions;         //事件机制的函数指针表
```

ngx_event_actions 我们已经在 14.4.6 节讨论过，它封装操作系统的底层事件模型，是 Nginx 事件处理机制的入口，Nginx 用它来添加、修改、删除和处理事件。

ngx_event_flags 是一个标志量，标记了当前 Nginx 事件机制的工作模式，由具体的 event 模块设置，取值是一系列的宏：

```
// 位于 event/nginx_event.h
#define NGX_USE_LEVEL_EVENT      0x00000001    //水平触发模式，即 LT
#define NGX_USE_ONESHOT_EVENT    0x00000002    //目前仅 kqueue 使用
#define NGX_USE_CLEAR_EVENT     0x00000004    //边缘触发模式，即 ET
#define NGX_USE_KQUEUE_EVENT     0x00000008    //使用 kqueue 系统调用
#define NGX_USE_GREEDY_EVENT    0x00000020    //使用非阻塞的“贪婪”模式
#define NGX_USE_EPOLL_EVENT     0x00000040    //使用 epoll 系统调用
...                                     //其他定义，如 poll、/dev/poll
```

Linux 下 Nginx 默认使用 epoll，它的值就是：

```
// 位于 event/modules/nginx_epoll_module.c
ngx_event_flags = NGX_USE_CLEAR_EVENT         //边缘触发模式，即 ET
                  |NGX_USE_GREEDY_EVENT        //使用非阻塞的“贪婪”模式
                  |NGX_USE_EPOLL_EVENT;        //使用 epoll 系统调用
```

14.7.3 负载均衡相关

在生产环境下通常都使用多个 worker 进程提供服务以充分利用多核 CPU，Nginx 内部实现了一种进程间的负载均衡功能，可以比较均匀地分配客户端连接，避免有的 worker 工作繁忙而有的 worker 却很“清闲”的状况出现。

负载均衡机制的实现依靠的是共享内存里的锁，它可以在进程间共享访问：

```
// 位于 event/nginx_event.c
ngx_atomic_t*      ngx_accept_mutex_ptr;       //负载均衡锁指针
ngx_shmtx_t        ngx_accept_mutex;          //负载均衡锁，在共享内存里
ngx_uint_t         ngx_use_accept_mutex;      //是否启用负载均衡
ngx_uint_t         ngx_accept_mutex_held;     //是否已持有负载均衡锁
ngx_msec_t         ngx_accept_mutex_delay;    //争抢负载均衡锁的等待时间
ngx_int_t          ngx_accept_disabled;      //暂停接受请求的计数
```

启用负载均衡机制的指令是“accept_mutex on”，它虽然可以让各个进程的负载更加均匀，但负面作用也非常明显，多个进程争抢锁浪费了 CPU 时间也降低了效率，所以目前 Nginx 官方不推荐使用负载均衡机制。

在 14.8.10 节将讨论 Nginx 的负载均衡工作原理。

14.7.4 统计相关

Nginx 支持简单的连接信息统计功能，同样也使用了共享内存：

```
// 位于 event/nginx_event.c
static ngx_atomic_t  connection_counter = 1;           //创建连接的计数器
ngx_atomic_t*        ngx_connection_counter = &connection_counter;
```

如果启用了模块 ngx_http_stub_status_module，那么还会有更多的统计信息：

```
// 位于 event/nginx_event.c
#if (NGX_STAT_STUB)                                //启用统计模块

static ngx_atomic_t  ngx_stat_accepted0;           //已接受的连接总数
ngx_atomic_t *ngx_stat_accepted = &ngx_stat_accepted0;
static ngx_atomic_t  ngx_stat_handled0;           //已处理的连接总数
ngx_atomic_t *ngx_stat_handled = &ngx_stat_handled0;
...                                                //其他信息

#endif
```

14.8 运行机制

之前我们已经全面介绍了 Nginx 事件机制的静态模型，包括 socket/epoll 系统调用、核心数据结构定义、定时器、event 模块体系和相关的全局变量，现在就可以正式开始剖析 Nginx 事件机制的工作流程。

14.8.1 模块初始化

ngx_event_core_module 总体管理 Nginx 事件机制，它的定义是：

```
ngx_module_t  ngx_event_core_module = {
    NGX_MODULE_V1,                                //标准填充宏
    &ngx_event_core_module_ctx,                    /* module context */
    ngx_event_core_commands,                       /* module directives */
    NGX_EVENT_MODULE,                             /* module type */
```

```

    NULL, /* init master */
    ngx_event_module_init, /* init module */
    ngx_event_process_init, /* init process */
    NULL, /* init thread */
    NULL, /* exit thread */
    NULL, /* exit process */
    NULL, /* exit master */
    NGX_MODULE_V1_PADDING //标准填充宏
};

```

可见，模块实现了两个初始化函数，分别是模块初始化和进程初始化：前者的执行时机是在 fork 之前的 ngx_init_cycle() 函数，初始化的数据会被后续的 worker 子进程完全复制共享；而后者是在 ngx_init_cycle() 函数和 fork 之后，每个进程单独运行不会共享。

Nginx 启动时执行 ngx_init_cycle()，读取配置文件，建立起整个模块架构并打开监听端口（参见 13.6 节）。对于 event 模块，除了创建配置结构体外，更重要的是执行 ngx_event_core_module 的 init_module 函数指针初始化进程共用的事件机制。

ngx_event_module_init() 做的工作不多，主要是创建共享内存，用于存放负载均衡锁和统计用的原子变量，这些都是为进程间通信准备的，代码摘要如下：

```

// 位于 event/nginx_event.c
cf = ngx_get_conf( /*获取 event 模块的配置结构体数组
    cycle->conf_ctx, ngx_events_module);
ecf = (*cf)[ngx_event_core_module.ctx_index]; //获取 event_core 模块配置

ccf = (ngx_core_conf_t *) ngx_get_conf( //获取 core 模块配置
    cycle->conf_ctx, ngx_core_module);

ngx_timer_resolution = ccf->timer_resolution; //设置时间精确度全局变量

getrlimit(RLIMIT_NOFILE, &rlmt); //系统调用，可打开的最多文件描述符

if (ccf->master == 0) { //如果是单进程模式，即只启动一个进程
    return NGX_OK; //那么结束，后续不使用负载均衡锁
}

shared = shm.addr; //shared 是共享内存的地址指针
ngx_accept_mutex_ptr = (ngx_atomic_t *) shared; //设置负载均衡锁

ngx_shmtx_create(&ngx_accept_mutex, //在共享内存里创建负载均衡锁
    (ngx_shmtx_sh_t *) shared, cycle->lock_file.data);

ngx_connection_counter = //共享内存里第二个位置是连接计数器

```

```
(ngx_atomic_t *) (shared + 1 * cl);
```

代码里需要注意的是对进程模式的判断, 如果使用单进程模式 (即 “master_process off”), 那么只会有一个 single 进程, 也就不必创建负载均衡相关的共享内存和锁。

14.8.2 进程初始化

在单进程模式的 ngx_single_process_cycle() 里, 或者多进程模式 fork 之后的每个 worker 进程的 ngx_worker_process_cycle() 里, 都会遍历模块数组, 调用模块的 init_process 函数指针, 执行每个模块的进程初始化代码。

ngx_event_core_module 的初始化函数是 ngx_event_process_init(), 它的工作非常重要, 创建了每个 worker 进程运行必需的事件机制数据结构, 主要内容是:

- 1) 决定是否启用负载均衡;
- 2) 初始化延后处理事件队列;
- 3) 初始化定时器红黑树;
- 4) 初始化使用的 event 模块, Linux 就是 epoll;
- 5) 如果设置了时间精确度, 那么调用 setitimer() 定时发送时间信号;
- 6) 创建连接池数组和对应的读写事件数组;
- 7) 初始化空闲连接链表;
- 8) 初始化监听端口, 为每个监听端口分配一个连接对象。

由于这些初始化操作都很重要, 所以下面我们分小节详细介绍。

14.8.3 基本参数初始化

事件机制的基本参数比较简单, 包括负载均衡锁、延后处理事件队列、定时器等, 代码摘要如下:

```
//多进程模式, 多于 1 个 worker 进程, 且启用了负载均衡
if (ccf->master && ccf->worker_processes > 1 && ecf->accept_mutex) {
    ngx_use_accept_mutex = 1;           //设置全局变量, 启用负载均衡
    ngx_accept_mutex_held = 0;          //刚开始未持有锁
    ngx_accept_mutex_delay = ecf->accept_mutex_delay; //设置争抢锁的等待时间
} else {
    ngx_use_accept_mutex = 0;           //单进程, 或未启用负载均衡, 则不使用锁
```

```

}

ngx_queue_init(&ngx_posted_accept_events); //初始化两个延后处理的事件队列
ngx_queue_init(&ngx_posted_events);

ngx_event_timer_init(cycle->log);           //初始化定时器红黑树

for (m = 0; cycle->modules[m]; m++) {        //遍历模块数组, 查找 event 模块
    ...                                     //根据 use 找到使用的 event 模块
    module->actions.init(cycle, ngx_timer_resolution); //模块初始化
    break;                                   //不会使用多于一个的 event 模块, 所以初始化后就 break 退出循环
}

setitimer(ITIMER_REAL, &itv, NULL);         //系统调用, 设置定时器信号

```

默认情况下 Nginx 不启用负载均衡, 相关的讲解见 14.8.11 节。

延后处理事件队列是标准的 `ngx_queue_t` 结构, 所以直接用函数宏 `ngx_queue_init` 初始化。

定时器红黑树的初始化使用的是 14.5.1 节的函数 `ngx_event_timer_init()`。

随后 Nginx 遍历模块数组, 根据 `ecf->use` 找到使用的 event 模块 (即使用的 I/O 多路复用技术), 调用模块的 `actions.init`, 也就是执行了具体 event 模块的初始化——Linux 系统里就是 `ngx_epoll_module`。

最后, 如果配置了指令 “`timer_resolution`”, Nginx 就使用 `setitimer()` 设置系统定时器, 让操作系统定时向本进程发送 `SIGALRM` 信号。

14.8.4 epoll 初始化

`ngx_epoll_module` 的初始化函数是 `ngx_epoll_init()`, 它是函数指针表 `actions` 里的 `init` 字段, 在 `ngx_event_core_module` 执行进程初始化时被调用, 初始化每个 worker 进程专用的 `epoll` 实例。

由于 `epoll` 的事件通知机制比较独立, 而且与 Nginx 多线程机制联系密切, 故本节暂不介绍, 关于它的详细内容见第 15 章。

`ngx_epoll_module` 使用三个静态全局变量保存 `epoll` 相关数据:

```

// 位于 event/modules/ngx_epoll_module.c
static int          ep = -1;           //epoll 文件描述符
static struct epoll_event* event_list; //epoll 事件数组, 用于 epoll_wait

```

```
static ngx_uint_t          nevents;          //epoll 事件数组的大小

函数 ngx_epoll_init() 首先调用 epoll_create() 创建 epoll 实例:

epcf = ngx_event_get_conf(cycle->conf_ctx, ngx_epoll_module);

if (ep == -1) {                                //epoll 未初始化
    ep = epoll_create(cycle->connection_n / 2); //系统调用创建 epoll 实例
}
```

之后创建 epoll 事件数组, 大小是 epcf->events, 用于 epoll_wait() 收集事件通知 (默认值是 512, 即一次最多处理 512 个事件), 注意这里没有使用 Nginx 的内存池, 而是直接使用系统内存:

```
if (nevents < epcf->events) {                    //检查是否改变了配置
    if (event_list) {                            //可能之前分配了内存空间
        ngx_free(event_list);                  //那么就释放内存
    }

    event_list = ngx_alloc(                      //分配内存, 创建数组
        sizeof(struct epoll_event) * epcf->events, cycle->log);
}

nevents = epcf->events;                        //设置数组的大小
```

函数最后初始化三个关键的事件机制全局变量: ngx_io、ngx_event_actions 和 ngx_event_flags, 它们的含义见 14.4.5、14.4.6 和 14.7.2 节:

```
ngx_io = ngx_os_io;                            //设置数据收发函数的全局访问点
ngx_event_actions =                            //设置事件处理函数的入口
    ngx_epoll_module_ctx.actions;              //使用 epoll 模块的 actions

ngx_event_flags = NGX_USE_CLEAR_EVENT        //边缘触发模式, 即 ET
    | NGX_USE_GREEDY_EVENT                    //使用非阻塞的“贪婪”模式
    | NGX_USE_EPOLL_EVENT;                    //使用 epoll 系统调用
```

ngx_epoll_init() 执行完之后, 事件处理函数和数据收发函数都已经准备好, Nginx 的事件机制可以说就初步建立了。

14.8.5 连接池初始化

每个 worker 进程都拥有独立的连接池和事件池, 也就是 ngx_cycle_t 里的成员 connections、read_events 和 write_events, 它们决定了单个进程所能处理的连接数量上限, 需要在进程启动时初始化:

```
// 位于 event/nginx_event.c
cycle->connections = //创建连接池数组, 大小是 connection_n
    ngx_alloc(sizeof(ngx_connection_t) * cycle->connection_n, cycle->log);

c = cycle->connections; //连接池数组的首地址

cycle->read_events = //创建读事件数组, 大小是 connection_n
    ngx_alloc(sizeof(ngx_event_t) * cycle->connection_n, cycle->log);

rev = cycle->read_events; //读事件数组的首地址
for (i = 0; i < cycle->connection_n; i++) { //初始化读事件数组
    rev[i].closed = 1; //读事件初始为关闭状态
    rev[i].instance = 1; //失效标志位置 1
}

cycle->write_events = //创建写事件数组, 大小是 connection_n
    ngx_alloc(sizeof(ngx_event_t) * cycle->connection_n, cycle->log);

wev = cycle->write_events; //写事件数组的首地址
for (i = 0; i < cycle->connection_n; i++) { //初始化写事件数组
    wev[i].closed = 1; //写事件初始为关闭状态
}
```

这段代码同样需要注意的是连接池和读写事件池数组的创建, 并没有使用任何内存池, 而是直接分配系统内存, 这意味着连接池对象的生命周期与 `ngx_cycle` 无关, 完全是属于进程自己的。

有了连接池和事件池之后, Nginx 开始关联读写事件并且初始化空闲连接链表:

```
// 位于 event/nginx_event.c
i = cycle->connection_n; //准备开始创建空闲链表
next = NULL; //链表的头指针, 初始为空

do { //注意 i 是数组的末尾, 从最后遍历
    i--; //i 递减至数组首个元素

    c[i].data = next; //设置连接对象的 next 指针
    c[i].read = &cycle->read_events[i]; //关联对应的读事件
    c[i].write = &cycle->write_events[i]; //关联对应的写事件
    c[i].fd = (ngx_socket_t) -1; //初始化连接, 无 socket

    next = &c[i]; //next 指针前移
} while (i); //循环直至 i 为 0, 即数组首位置

cycle->free_connections = next; //设置空闲连接链表头指针
```

```
cycle->free_connection_n = cycle->connection_n; //空闲连接链表长度
```

在初始化时连接池里的所有对象都是未使用的空闲状态，所以 Nginx 利用了连接对象 `ngx_connection_t` 的 `data` 字段作为链表的后继指针，把数组里连接对象串成了一个单向链表，遍历的同时也为连接对象关联了读写事件。

这样，进程里的连接池就完成了初始化，总共有 `connection_n` 个可用的连接对象，每个连接对象都可以监控读写事件——也就是收发数据，连接对象的使用则直接从链表 `free_connections` 里摘取即可，非常简单高效。

14.8.6 监听端口初始化

完整地描述 Nginx 启动监听的过程需要结合 `http` 或 `stream` 模块讲解，这里我们仅关注监听的事件处理部分。

如果在 `http{}`、`stream{}` 等配置里使用了 `listen` 指令，那么 `cycle->listening` 里会保存有这些监听端口的信息，并且在 `ngx_init_cycle()` 里用 `bind()` / `listen()` 启动监听，但因为这时还没有启动事件机制（`epoll`、连接池），所以并不能 `accept` 客户端请求。

`fork` 之后，每个 `worker` 进程都继承了 `master` 进程里打开的监听端口，就可以从连接池里获取空闲连接对象，关联到监听端口，把读事件加入 `epoll` 里从而开始服务：

```
// 位于 event/ngx_event.c
ls = cycle->listening.elts; //所有的监听端口数组
for (i = 0; i < cycle->listening.nelts; i++) { //遍历监听端口数组

    c = ngx_get_connection(ls[i].fd, cycle->log); //获取一个空闲连接对象

    c->type = ls[i].type; //设置连接的类型，TCP|UDP
    c->listening = &ls[i]; //连接对象关联到监听对象
    ls[i].connection = c; //监听对象关联到连接对象

    rev = c->read; //监听端口只关心读事件
    rev->accept = 1; //监听必须置 accept 标志为 1
    rev->handler = (c->type == SOCK_STREAM) ? //重要，设置读事件的回调函数
        ngx_event_accept : ngx_event_recvmsg; //TCP、UDP 的回调不同

    if (ngx_use_accept_mutex) { //如果使用负载均衡，那么初始化完毕
        continue; //继续处理下一个监听端口
    }

    ngx_add_event(rev, NGX_READ_EVENT, 0); //否则 LT 模式加入 epoll，立即开始服务
}
```


监听对象必须要配合一个连接对象才能工作,所以它首先要从连接池里获取一个空闲连接对象,关联到它的 `connection` 字段。

因为监听端口只有读事件而没有写事件^①,所以只需要设置读事件的回调函数,也就是当客户端连接成功,即将 `accept` 时要执行的函数。早期 Nginx 只支持 TCP,使用的回调函数是 `ngx_event_accept`,自 1.9.x 后 Nginx 开始支持 UDP,所以新增了一个函数 `ngx_event_recvmmsg`,两者的区分就依靠连接的协议类型。

如果不使用负载均衡(这是 Nginx 的默认设置),那么读事件就会立刻加入事件监控机制(`epoll`、`kqueue` 等),这样 `worker` 进程就可以立刻得到操作系统的事件通知,调用 `ngx_event_accept` 或 `ngx_event_recvmmsg` 处理客户端的连接,继而调用 `ls->handler` 执行 `http` 或 `stream` 处理(见 14.8.10 节)。

如果启用了负载均衡,那么进程接下来就要开始争抢锁的流程,只有抢到锁的进程才有资格把监听端口的读事件加入监控机制接受客户端连接,其他未抢到锁的进程只能等待下一次机会,等待的时间里(即 `ngx_accept_mutex_delay`)只能处理非 `accept` 的其他事件。

我们需要注意 Nginx 在添加读事件的监控时传递的参数是 0,也就是 `NGX_LEVEL_EVENT`(见 14.8.7 节),即以 LT 的模式加入 `epoll` 监控的,所以对于监听事件即使没有及时处理也不要紧,下一次 `epoll_wait()` 调用还是能够得到这个事件的通知,绝对不会丢失客户端的连接。

至此,Nginx 事件机制的初始化就全部结束,`worker` 进程进入无限循环,正式开始处理网络事件和定时器事件,对外提供服务。

14.8.7 初始化流程图

Nginx 事件机制的初始化工作比较多,采用图表的方式可以更清晰地从整体上掌握理解,如图 14-7 所示。

① 个人觉得 Nginx 监听端口没有用到写事件,对连接对象的使用有点儿“浪费”,也许可以单独为监听端口创建一个专用的连接池。

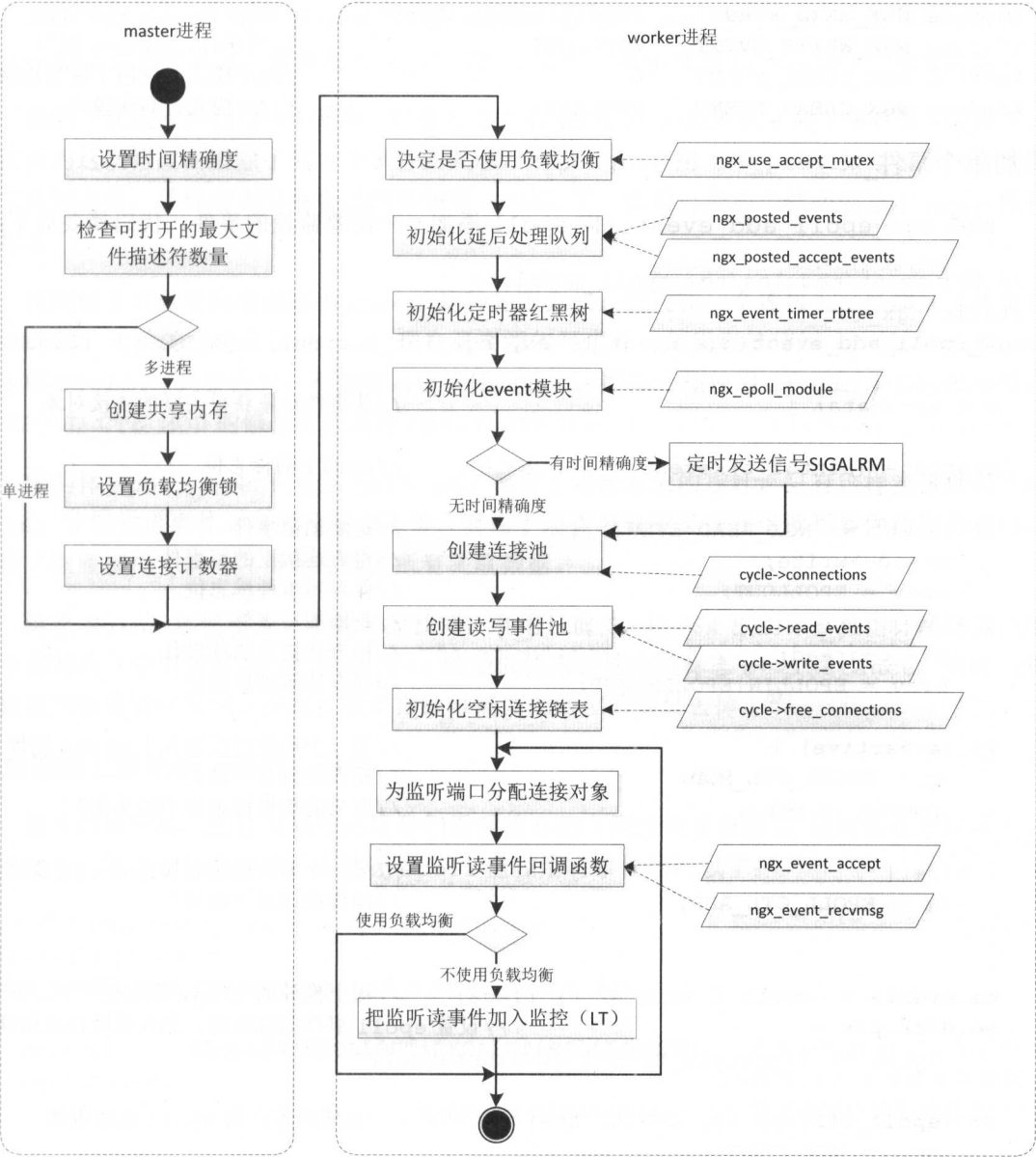


图 14-7 Nginx 事件机制的初始化流程

14.8.8 添加事件

Nginx 定义了一些宏来屏蔽操作系统事件处理机制的差异，对于 `epoll` 是：

```
// 位于 event/nginx_event.h
```

```

#define NGX_READ_EVENT      (EPOLLIN|EPOLLRDHUP)    //读事件, socket 可读
#define NGX_WRITE_EVENT     EPOLLOUT                //写事件, socket 可写
#define NGX_LEVEL_EVENT     0                       //LT 模式, 仅用于接受连接
#define NGX_CLEAR_EVENT     EPOLLET                 //ET 模式, 高速模式

```

添加单个事件

函数 `ngx_epoll_add_event()` 向 `epoll` 添加一个需要监控的事件, 代码摘要如下:

```

// 位于 event/modules/nginx_epoll_module.c
static ngx_int_t
ngx_epoll_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
{
    c = ev->data;                                //从事件对象获得关联的连接对象

    events = (uint32_t) event;                    //epoll 的标志位

    if (event == NGX_READ_EVENT) {                //是添加读事件
        e = c->write;                              //检查连接上的写事件
        prev = EPOLLOUT;                          //保存写事件标志位
    } else {                                       //是添加写事件
        e = c->read;                                //检查连接上的读事件
        prev = EPOLLIN|EPOLLRDHUP;                //保存读事件标志位
    }

    if (e->active) {                              //另一个事件已经加入了 epoll 监控
        op = EPOLL_CTL_MOD;                       //操作应该是“修改”
        events |= prev;                           //原来的事件标志位不能丢失!
    } else {                                       //另一个事件还没有加入 epoll 监控
        op = EPOLL_CTL_ADD;                       //操作应该是“新增”
    }

    ee.events = events | (uint32_t) flags;         //加上额外的 flags 标志位
    ee.data.ptr =                                //设置 epoll 事件关联数据, 加入失效标志位信息
        (void *) ((uintptr_t) c | ev->instance);

    if (epoll_ctl(ep, op, c->fd, &ee) == -1) {    //系统调用, 向 epoll 添加事件
        return NGX_ERROR;
    }

    ev->active = 1;                               //添加完毕, 设置事件活跃标志位
    return NGX_OK;
}

```

函数的基本操作是系统调用 `epoll_ctl()`, 但在执行之前必须要决定合适的参数。

入口参数 `event` 必须是 `NGX_READ_EVENT` 或 `NGX_WRITE_EVENT`，表示读写事件，`flags` 则是附加标志位，通常是 `NGX_CLEAR_EVENT`，表示使用 `epoll` 的 `ET` 高速模式。

虽然一次操作只修改一个事件，但事件对应的 `socket` 上是关联了两个事件的（读和写），不能因为这次修改而删除了另一个事件的关联，所以 `Nginx` 用 `e` 和 `prev` 保存了另一个事件和它的标志位，并检查事件是否已经添加到 `epoll` 了，如果已经添加就必须补上 `prev` 保存的标志。

使用例子可以更好地理解 `Nginx` 的这种处理方式。假设在一个连接上已经添加了读事件 `EPOLLIN`，事件的状态是 `active`，现在又要添加写事件，如果直接使用 `EPOLLOUT` 那么连接上的 `EPOLLIN` 就会丢失，不能收到读事件的通知，所以必须使用 `EPOLLIN|EPOLLOUT`，以 `EPOLL_CTL_MOD` 的修改方式调用 `epoll_ctl()`。

结构体 `epoll_event` 里的 `data.ptr` 保存了连接对象的指针，只有这样当事件发生时 `Nginx` 才能知道事件对应的是哪个连接，而由于连接对象里又关联了读写事件和监听端口，所以“顺藤摸瓜”就可以得到连接的所有关联数据。

这里 `Nginx` 还使用了一个特别的技巧，利用目前 32 位/64 位的计算机指针地址低位都是 0 的特性（字节对齐），用 `data.ptr` 的最低位来存储失效标志 `e->instance`，事件发生时比较判断是否有变化，在真正取出指针时需要把低位的信息去掉。^①

添加连接上的所有（读写）事件

很多时候连接上的读写事件都是我们需要关心的（同时收发数据），这时添加事件就没有那么复杂，不需要什么参数，直接让 `epoll` 监控所有事件就可以了：

```
// 位于 event/modules/nginx_epoll_module.c
static ngx_int_t
ngx_epoll_add_connection(ngx_connection_t *c)
{
    ee.events = EPOLLIN|EPOLLOUT|EPOLLET|EPOLLRDHUP; //直接是所有 epoll 标志位
    ee.data.ptr = //设置 epoll 事件关联数据
        (void *) ((uintptr_t) c | c->read->instance); //使用读事件的失效标志位

    if (epoll_ctl(ep, EPOLL_CTL_ADD, c->fd, &ee) == -1) { //向 epoll 添加事件
        return NGX_ERROR;
    }
}
```

① 这也是个“不得已”的技巧，因为结构体 `epoll_event` 里可以存放的数据实在是太有限了，`Nginx` 只能把连接的信息以非常规的方式“挤”进 `data.ptr` 这一个字段里，如果 `epoll_event` 能够再多点空间相信 `Nginx` 也不会用这种实现手法。

```

    }

    c->read->active = 1;                //添加完毕
    c->write->active = 1;                //读写事件都是活跃的

    return NGX_OK;
}

```

添加事件的便捷操作

`ngx_epoll_add_event()` 和 `ngx_epoll_add_connection()` 比较底层，最好不要直接调用，所以 Nginx 又定义了两个简化的函数，它们也同时屏蔽了操作系统的差异：

```

// 位于 event/nginx_event.c
ngx_int_t
ngx_handle_read_event(ngx_event_t *rev, ngx_uint_t flags)
{
    if (ngx_event_flags & NGX_USE_CLEAR_EVENT) { //使用 epoll 的 ET 模式

        if (!rev->active && !rev->ready) {          //已添加过则不再重复添加
            if (ngx_add_event(                      //添加读事件，使用 ET
                rev, NGX_READ_EVENT, NGX_CLEAR_EVENT) == NGX_ERROR)
            {
                return NGX_ERROR;
            }
        }
        return NGX_OK;
    }
    return NGX_OK;
}

ngx_int_t
ngx_handle_write_event(ngx_event_t *wev, size_t lowat)
{
    if (ngx_event_flags & NGX_USE_CLEAR_EVENT) { //使用 epoll 的 ET 模式

        if (!wev->active && !wev->ready) {          //已添加过则不再重复添加
            if (ngx_add_event(wev, NGX_WRITE_EVENT, //添加写事件，使用 ET
                NGX_CLEAR_EVENT | (lowat ? NGX_LOWAT_EVENT : 0))
                == NGX_ERROR)
            {
                return NGX_ERROR;
            }
        }
        return NGX_OK;
    }
    return NGX_OK;
}

```

这两个函数在调用 `ngx_add_event()` 时都使用了 `NGX_CLEAR_EVENT` 标志位，也就是

EPOLLET，所以不需要再“画蛇添足”，Nginx 会默认使用高效的 ET 模式。

在我们自己编写程序时应该尽量使用 `ngx_handle_read_event()` 和 `ngx_handle_write_event()` 这两个函数，`flags` 和 `lowat` 参数都用 0（因为 Linux 不支持 `lowat`）。

14.8.9 删除事件

Nginx 删除事件也有对单个事件的操作和对连接的操作两种，不过没有简化操作函数。

删除单个事件

函数 `ngx_epoll_del_event()` 在 `epoll` 里删除一个事件（读或写），操作与 `ngx_epoll_add_event()` 有些类似，也要防止删除对另一个事件的监控，但逻辑要略微简单一些，代码摘要如下：

```
// 位于 event/modules/ngx_epoll_module.c
static ngx_int_t
ngx_epoll_del_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
{
    c = ev->data;                                //从事件对象获得关联的连接对象

    if (event == NGX_READ_EVENT) {                //要删除读事件
        e = c->write;                               //检查连接上的写事件
        prev = EPOLLOUT;                           //保存写事件标志位
    } else {                                        //要删除写事件
        e = c->read;                                //检查连接上的读事件
        prev = EPOLLIN|EPOLLRDHUP;                 //保存读事件标志位
    }

    if (e->active) {                                //另一个事件已经加入了 epoll 监控
        op = EPOLL_CTL_MOD;                         //操作应该是“修改”
        ee.events = prev | (uint32_t) flags;        //原来的事件标志位不能丢失！
        ee.data.ptr =                               //设置 epoll 事件关联数据，加入失效标志位信息
            (void *) ((uintptr_t) c | ev->instance);
    } else {                                        //另一个事件还没有加入 epoll 监控
        op = EPOLL_CTL_DEL;                         //操作应该是“删除”
        ee.events = 0;                               //直接清空所有 epoll 标志位
        ee.data.ptr = NULL;                         //也不需要保存额外数据
    }

    if (epoll_ctl(ep, op, c->fd, &ee) == -1) { //系统调用，在 epoll 里删除事件
        return NGX_ERROR;
    }

    ev->active = 0;                                //删除完毕，清除事件活跃标志位
    return NGX_OK;
}
```

```
}
```

删除连接上的所有（读写）事件

函数 `ngx_epoll_del_connection()` 直接删除在 `epoll` 里监控的读写事件，并置事件为不活跃状态：

```
// 位于 event/modules/nginx_epoll_module.c
static ngx_int_t
ngx_epoll_del_connection(ngx_connection_t *c, ngx_uint_t flags)
{
    op = EPOLL_CTL_DEL;                // 直接是删除操作
    ee.events = 0;                      // 直接清空所有 epoll 标志位
    ee.data.ptr = NULL;                 // 也不需要保存额外数据

    if (epoll_ctl(ep, op, c->fd, &ee) == -1) { // 系统调用，在 epoll 里删除事件
        return NGX_ERROR;
    }

    c->read->active = 0;                // 删除完毕
    c->write->active = 0;               // 读写事件都不再活跃

    return NGX_OK;
}
```

14.8.10 处理事件

`ngx_process_events_and_timers()` 是 Nginx 事件机制的核心函数，也是 worker 进程里无限循环的主体（见 13.8.1 和 13.9.4 节），让我们回忆一下 Nginx worker 进程里的代码：

```
// 位于 os/unix/nginx_process_cycle.c
for ( ;; ) {                                // 进程的无限循环，处理事件和信号
    ngx_process_events_and_timers(cycle);    // 处理网络事件和定时器事件
    ...                                     // 处理 UNIX 信号
}
```

可见，worker 进程在运行时就是周而复始地执行这个函数，收集各种网络连接事件和定时器事件，把事件分发给 `http`、`stream` 等模块处理。

ngx_process_events_and_timers

如果不使用负载均衡，那么 `ngx_process_events_and_timers()` 的逻辑就很简单，如它的名字一样，处理网络事件和定时器事件（本节的代码均省略了负载均衡和延后处理队列

相关的部分):

```
// 位于 event/nginx_event.c
void ngx_process_events_and_timers(ngx_cycle_t *cycle)
{
    if (ngx_timer_resolution) {                //要求时间精确度
        timer = NGX_TIMER_INFINITE;           //使用-1, 要求 epoll_wait 无限等待
        flags = 0;                             //直至收到 SIGALRM 信号
    } else {                                    //不要求时间精确度
        timer = ngx_event_find_timer();        //在定时器红黑树里找到即将超时的事件
        flags = NGX_UPDATE_TIME;              //使用这个时间作为 epoll_wait 等待时间
    }

    delta = ngx_current_msec;                  //记录当前的毫秒数
    (void) ngx_process_events(cycle, timer, flags); //收集事件并逐个处理
    delta = ngx_current_msec - delta;          //计算消耗的时间

    if (delta) {                                //如果消耗了一些时间, 那么 delta>0
        ngx_event_expire_timers();            //查找超时事件并逐个处理
    }
}
```

函数先要决定 `epoll_wait()` 阻塞等待的时间参数。如果配置了时间精确度 (指令 “`timer_resolution`”), 那么就可以让 `epoll` 一直阻塞等待, 由系统发送信号 `SIGALRM` 来中断。否则就需要在定时器红黑树里找到即将超时的事件, 用这个时间作为阻塞的时间。如果红黑树是空的, 那么 `ngx_event_find_timer()` 也会返回 `NGX_TIMER_INFINITE`, `epoll` 会一直阻塞等待事件的发生, 但并不会会有信号 `SIGALRM` 中断。

之后的 `ngx_process_events()` 是 Nginx 的事件处理的主逻辑, 也就是 `epoll` 模块的 `ngx_epoll_process_events()`, 调用 `epoll_wait()` 处理事件, 并更新时间。

变量 `delta` 用来度量事件处理消耗的时间, 但 Nginx 并不关心具体的值。只要它大于 0, 就说明已经流逝了一些时间, 需要检查定时器红黑树, 找出可能已经超时的事件并处理。

ngx_epoll_process_events

`epoll` 模块处理事件的函数是 `ngx_epoll_process_events()`, 它调用 `epoll_wait()`, 把收集到的事件存储在全局变量 `event_list` 里:

```
// 位于 event/modules/nginx_epoll_module.c
events = epoll_wait(ep, event_list,           //系统调用收集事件
                    (int) nevents, timer);    //最多 nevents 个, 实际 events 个

if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) { //要求更新时间
```



```
    ngx_time_update(); //调用 gettimeofday() 更新时间
}
```

epoll_wait() 阻塞等待的时间由 ngx_process_events_and_timers() 传递过来的 timer 参数确定, 可以是确定的时间, 也可以是无限等待, 直至有事件发生或者收到 SIGALRM, 这时 Nginx 就会调用 gettimeofday() 更新时间。

通常情况下 Nginx 都很繁忙, 所以 epoll_wait() 基本不会阻塞等待而是立即返回, 这时更新时间就可以得到比 ngx_timer_resolution 更精细的粒度。

从系统内核得到事件存储在数组 event_list 里, 共 events 个, Nginx 需要顺序处理这些事件, 执行它们的 handler 函数指针, 这就是所谓的“事件驱动”:

```
for (i = 0; i < events; i++) { //遍历收集到的事件
    c = event_list[i].data.ptr; //取事件关联的数据, 是连接对象指针

    instance = (uintptr_t) c & 1; //取最低位保存的失效标志位
    c = (ngx_connection_t *) ( //忽略低位得到正确的指针
        (uintptr_t) c & (uintptr_t) ~1);

    rev = c->read; //检查连接上的读事件
    if (c->fd == -1 || rev->instance != instance) { //连接已失效
        continue; //不做处理, 直接看下一个事件
    }

    revents = event_list[i].events; //取 epoll 事件标志位
    if (revents & (EPOLLERR|EPOLLHUP)) { //如果发生了错误或者客户端断连
        revents |= EPOLLIN|EPOLLOUT; //加上读写标志, 方便后续代码处理
    }

    if ((revents & EPOLLIN) && rev->active) { //是读事件且事件 active
        rev->ready = 1; //设置读事件 ready, 即可读
        rev->handler(rev); //立即执行回调函数, 处理事件
    } //读事件处理结束

    wev = c->write; //检查连接上的写事件
    if ((revents & EPOLLOUT) && wev->active) { //是写事件且事件 active
        if (c->fd == -1 || wev->instance != instance) { //连接已失效
            continue; //不做处理, 直接看下一个事件
        }
        wev->ready = 1; //设置写事件 ready, 即可写
        wev->handler(wev); //立即执行回调函数, 处理事件
    } //写事件处理结束
} //for 循环结束
```

Nginx 先从 `data.ptr` 里取出之前添加事件时存储的连接对象指针 (14.8.6 节), 因为最低位被用来存储失效标志位 `instance`, 所以必须要用位操作把它们分离开。

事件只有读和写两种, 但读事件里又有需要优先处理的 `accept` 事件, 所以 Nginx 先检查读事件。如果事件的 `instance` 标志位与连接里取出的 `instance` 不一致, 那么可能是在等待事件发生的过程中连接被关闭后被重用了, 也就意味着这个事件已经没有意义, 所以不需要处理, 直接忽略。

如果读事件的标志位是 `EPOLLIN` 或 `EPOLLHUP`, 说明连接上有数据可读, 就可以置 `ready` 标志位, 然后立即调用 `rev->handler` 处理这个事件。同样的, 如果写事件的标志位是 `EPOLLOUT`, 说明连接可写, 同样置 `ready` 标志位后立即调用 `wev->handler` 处理。

就这样反复循环, 直至数组 `event_list` 里的所有事件处理完毕, 完成一次 `epoll_wait` 事件处理, 然后再由 `worker` 进程的 `for(;;)` 通过 `ngx_process_events_and_timers()` 再次调用, 不停地收集 `epoll` 事件通知并处理。

流程图

`ngx_process_events_and_timers()` 的处理流程如图 14-8 所示。

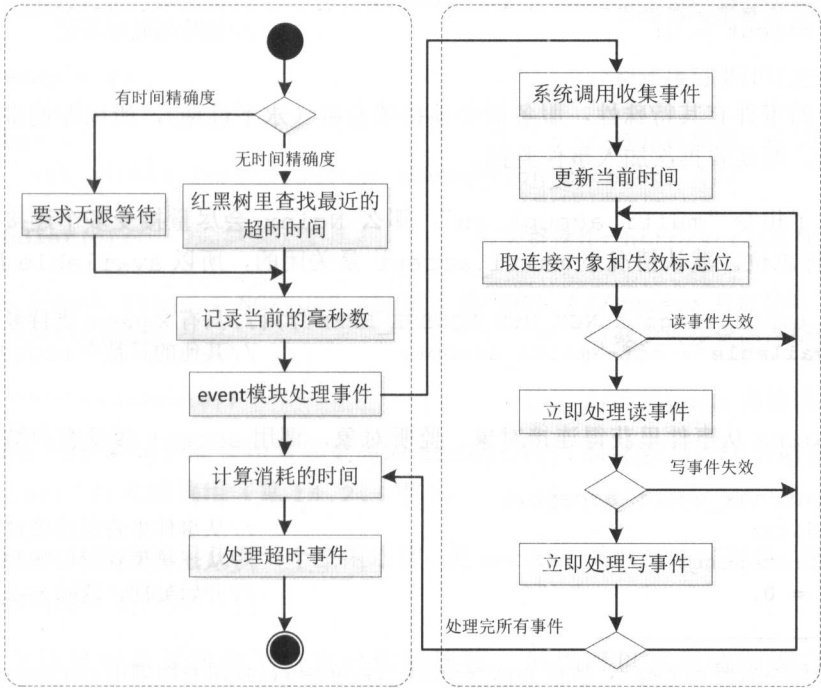


图 14-8 `ngx_process_events_and_timers()` 的处理流程

14.8.11 接受连接

worker 进程在初始化时把监听端口的读事件加入了 `epoll` 监控，并设置了读事件的回调函数（14.8.5 节）：

```
// 位于 event/nginx_event.c
rev->handler = (c->type == SOCK_STREAM) ?      //重要，设置读事件的回调函数
    ngx_event_accept : ngx_event_recvmsg;      //TCP、UDP 的回调不同
```

这就意味着当客户端与服务器成功建立连接之后，`ngx_process_events_and_timers()` 里会收到内核的 `epoll` 通知，在监听端口关联的连接对象上有读事件发生，TCP/UDP 协议分别使用 `ngx_event_accept()` 和 `ngx_event_recvmsg()` 来处理。

工作流程

这两个函数的实现很类似，所以我们只讨论 `ngx_event_accept()`。^①

函数首先检查事件是否超时：

```
// 位于 event/nginx_event_accept.c
if (ev->timedout) {                                //监听事件是否已经超时了
    ngx_enable_accept_events(ngx_cycle);           //如果超时就再次加入 epoll 监控
    ev->timedout = 0;                               //清除超时标志位
}
```

监听端口的事件有其特殊性，服务器必须持续监听（永不过期），所以即使事件超时也不能认为它出错，而是要再次加入事件监控。

如果配置了指令“`multi_accept on`”，那么 Nginx 会尽量接受多个连接，所以设置 `available` 标志位。默认情况下 `multi_accept` 是关闭的，所以 `available` 是 0。^②

```
if (!(ngx_event_flags & NGX_USE_KQUEUE_EVENT)) { //只有 kqueue 支持多 accept
    ev->available = ecf->multi_accept;             //其他的只是个 bool 标志位
}
```

接下来 Nginx 从事件里获得连接对象、监听对象，调用 `accept` 接受客户端的连接：

```
// 位于 event/nginx_event_accept.c
lc = ev->data;                                     //从事件里获得连接对象
ls = lc->listening;                                //从连接里获得监听对象
ev->ready = 0;                                     //开始处理，清除 ready 标志位
```

① 因为 UDP 协议不需要 `accept`，所以 `ngx_event_recvmsg()` 使用系统调用 `recvmsg()` 来代替。

② `multi_accept` 仅对 FreeBSD 的 `kqueue` 机制有意义，对于 Linux 的 `epoll` 效果并不大。

```

do {
    s = accept4(lc->fd,
                &sa.sockaddr, &socklen, SOCK_NONBLOCK);
    //对于非 kqueue 此循环无意义
    //从内核获取一个客户端连接

    ngx_accept_disabled =
        ngx_cycle->connection_n / 8 - ngx_cycle->free_connection_n;
    //计算空闲连接数

    c = ngx_get_connection(s, ev->log);
    //分配一个空闲连接对象

    c->type = SOCK_STREAM;
    //连接的类型设置为 TCP
    c->pool = ngx_create_pool(ls->pool_size);
    //创建连接使用的内存池
    ngx_memcpy(c->sockaddr, &sa, socklen);
    //拷贝客户端地址到连接对象

    c->recv = ngx_recv;
    //设置连接的接收函数指针
    c->send = ngx_send;
    //设置连接的发送函数指针
    c->recv_chain = ngx_recv_chain;
    //设置连接的接收函数指针
    c->send_chain = ngx_send_chain;
    //设置连接的发送函数指针
    c->listening = ls;
    //关联到监听端口
    c->local_sockaddr = ls->sockaddr;
    //保存服务器地址

    rev = c->read;
    //连接的读事件对象
    wev = c->write;
    //连接的写事件对象

    wev->ready = 1;
    //新连接写事件肯定是 ready 的

    c->number =
        ngx_atomic_fetch_add(ngx_connection_counter, 1);
    //连接计数器, 标记用的序号

    ls->handler(c);
    //关键操作, http/stream 处理机制的入口!

    if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) { //kqueue 专有逻辑
        ev->available--;
        //减少可接受的连接数
    }
} while (ev->available);
//非 kqueue 直接退出循环

```

在 Linux 系统下 Nginx 使用效率更高的 `accept4()`, 直接获得一个非阻塞的 `socket`, 减少了一次 `ioctl()` 系统调用 (见 14.2.4 节)。

随后 Nginx 从连接池里获取一个空闲连接, 把 `socket` 关联到连接对象, 设置协议类型、地址、收发函数指针等字段。

现在这个连接对象就代表了与客户端的连接, 所以可以调用监听端口的回调函数 `handler`, 正式开始与客户端的通信。对于 `http` 模块, 执行的是 `ngx_http_init_`

connection(), 对于 stream 模块, 执行的是 ngx_stream_init_connection(), 分别进入 http 或 stream 框架处理。

如果启用了 multi_accept, 那么 ev->available 就是 1, Nginx 会继续循环, 继续在监听端口上接受请求创建连接对象, 直至 accept 出错退出循环。如果关闭 multi_accept (Linux 下通常都是关闭), 那么 Nginx 本次只接受一个客户端连接, do-while 循环没有任何意义。

在接受连接时 Nginx 还计算了全局变量 ngx_accept_disabled, 它用来控制负载均衡, 详细解释见 14.8.11 节。

流程图

ngx_event_accept() 接受客户端请求的工作流程如图 14-9 所示。

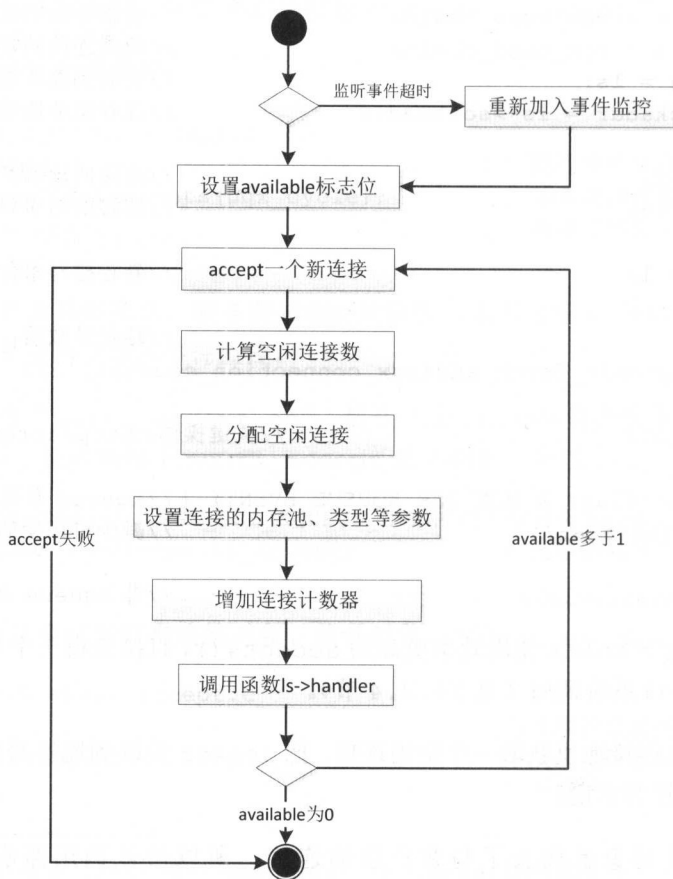


图 14-9 ngx_event_accept() 接受客户端请求的工作流程

14.8.12 负载均衡

早期 Nginx 引以为傲的一项功能就是内置的“负载均衡”，可以不依赖操作系统在各个 worker 进程之间较均匀地处理连接，充分利用 CPU。但多年的实践证明，虽然 Nginx 的负载均衡机制出发点很好，设计的也很精妙，但却“拖累”了整个服务器的性能，对于高速 CPU、繁忙的系统更是明显。

所以，自 1.11.3 开始，Nginx 终于默认关闭了负载均衡功能，相当于变相官方宣布了负载均衡的“死亡”。

不过对于我们来说，研究 Nginx 负载均衡机制的工作原理还是很有意义的，可以学习它的设计思想和实现方式，提高我们自身的开发水平。

启用负载均衡

启用负载均衡机制的指令是“accept_mutex on”，在进程初始化时会设置相关的全局变量（14.8.3 节）：

```
//多进程模式，多于1个worker进程，且启用了负载均衡
if (ccf->master && ccf->worker_processes > 1 && ecf->accept_mutex) {
    ngx_use_accept_mutex = 1;           //设置全局变量，启用负载均衡
    ngx_accept_mutex_held = 0;          //刚开始未持有锁
    ngx_accept_mutex_delay = ecf->accept_mutex_delay; //设置争抢锁的等待时间
}
```

启用负载均衡必须同时满足三个条件：多进程模式，多个 worker 进程，并且显式启用负载均衡（“accept_mutex on”）。这时 Nginx 就会设置负载均衡使用的全局变量，首先是启用标志 ngx_use_accept_mutex 置 1。此时进程还没有开始服务，也没有争抢锁，所以 ngx_accept_mutex_held 是 0。因为不能频繁地抢锁，所以未抢到锁的进程要略做等待，等待时间最多是 ngx_accept_mutex_delay，由指令“accept_mutex_delay”确定。

争抢 accept 锁

在进程初始化的最后（14.8.5 节），如果启用了负载均衡，那么 worker 进程不会立即把监听端口加入 epoll，只有调用 ngx_trylock_accept_mutex() 抢到了锁的进程才“有资格”从监听端口获取读事件接受连接，它是 Nginx 负载均衡实现的关键。

accept 锁的持有者是唯一的，所以任一时刻监听端口只有唯一一个 worker 进程：

```
// 位于 event/nginx_event_accept.c
ngx_int_t ngx_trylock_accept_mutex(ngx_cycle_t *cycle)
{
```

```

if (ngx_shmtx_trylock(&ngx_accept_mutex)) { //无阻塞抢共享锁

    if (ngx_accept_mutex_held) { //之前已经持有锁
        return NGX_OK; //不需要添加，可以继续接受连接
    }

    ngx_enable_accept_events(cycle); //把监听端口的读事件加入监控
    ngx_accept_mutex_held = 1; //设置持有锁的标志量
    return NGX_OK;
}

if (ngx_accept_mutex_held) { //没抢到锁，但之前持有锁
    ngx_disable_accept_events(cycle, 0); //删除监听端口的读事件监控
    ngx_accept_mutex_held = 0; //清除持有锁的标志量
}

return NGX_OK; //没抢到锁，不能接受新连接
}

```

ngx_trylock_accept_mutex() 使用存放在共享内存里的 ngx_accept_mutex 作为负载均衡锁，抢锁的过程是非阻塞的。

如果抢到了锁，那么就调用 ngx_enable_accept_events() 把 cycle->listening 里的所有监听端口加入 epoll 的监控，这样在调用 epoll_wait() 时此进程就能够获得内核建立连接事件的通知，使用 accept 建立新连接。

如果没有抢到锁，Nginx 就检查 ngx_accept_mutex_held 标志量，必须删除之前监控的读事件，避免出现两个以上的进程同时监控的情况出现（即所谓的“惊群”）。

负载均衡的连接数控制

全局变量 ngx_accept_disabled 是 Nginx 负载均衡的一个重要参数，决定 worker 进程能否去抢锁接受新连接，它的计算在函数 ngx_event_accept() 里：

```

// 位于 event/nginx_event_accept.c
ngx_accept_disabled = //计算空闲连接数
    ngx_cycle->connection_n / 8 - ngx_cycle->free_connection_n;

ngx_accept_disabled 的值是总连接数的 1/8 减去当前的空闲连接数。

```

如果空闲连接足够多，那么它就是负值。如果 Nginx 很繁忙，消耗了大量的可用连接在处理网络收发，那么空闲连接就会逐渐减少，ngx_accept_disabled 相应地就会逐渐增加。当空闲连接不足总连接数的 1/8 时 ngx_accept_disabled 就变成了正值。

收集事件

在事件机制处理事件的函数 `ngx_process_events_and_timers()` 里，如果启用了负载均衡就会检查 `ngx_accept_disable`，决定是否争抢锁 `accept` 连接：

```
// 位于 event/nginx_event.c
if (ngx_use_accept_mutex) {
    if (ngx_accept_disabled > 0) {
        ngx_accept_disabled--;
    } else {
        ngx_trylock_accept_mutex(cycle);

        if (ngx_accept_mutex_held) {
            flags |= NGX_POST_EVENTS;
        } else {
            if (timer == NGX_TIMER_INFINITE
                || timer > ngx_accept_mutex_delay)
            {
                timer = ngx_accept_mutex_delay;
            }
        }
    }
}
```

//启用负载均衡功能
//工作繁忙，空闲连接不足
//减1，缓慢提升优先级

//还有较多的空闲连接
//尝试抢锁

//成功抢到了锁，开始接受新连接
//置标志位，要求延后处理事件
//没抢到锁，等待下一次机会
//检查等待的时间
//不能无限或者过长时间等待

//最多等待指定的时间

//判断空闲连接的 if 结束
//负载均衡逻辑结束

`(void) ngx_process_events(cycle, timer, flags);` //收集处理事件，更新时间

如果 `ngx_accept_disable` 大于 0，意味着此时 Nginx 的可用连接已经很紧张了（剩余不到 12.5%），再接受更多的新连接将很快导致连接池耗尽，这时 worker 进程可以进入负载均衡状态，不去抢锁，即暂时不接受新连接，只处理之前已经建立的网络连接，避免过载，把 `accept` 新连接的机会让给其他的 worker 进程。

如果 `ngx_accept_disable` 小于 0，意味着进程里还有足够的空闲连接，就可以调用 `ngx_trylock_accept_mutex()` 与其他进程竞争抢锁。抢成功就可以 `accept` 新连接，抢失败则只能等待，但等待的时间不能超过 `ngx_accept_mutex_delay`。

为了避免长期持有锁导致其他进程抢不到，Nginx 使用了标志位 `NGX_POST_EVENTS`，要求函数 `ngx_process_events()` 不立即处理事件，而是把事件放入延后队列，因为队列的入队操作是非常快的，所以 `ngx_process_events()` 可以在瞬间“处理”完所有 `epoll_wait()` 收集到的事件：

```
// 位于 event/modules/nginx_epoll_module.c
if ((revents & EPOLLIN) && rev->active) { //是读事件且事件 active
```



```

    if (flags & NGX_POST_EVENTS) { //要求延后处理事件
        queue = rev->accept ? //由 accept 标志位决定使用的队列
            &ngx_posted_accept_events: &ngx_posted_events;
        ngx_post_event(rev, queue); //放入队列，暂时不处理
    } else { //不要求延后处理事件
        rev->handler(rev); //立即执行回调函数，处理事件
    }
} //读事件处理结束

if ((revents & EPOLLOUT) && wev->active) { //是写事件且事件 active

    if (flags & NGX_POST_EVENTS) { //要求延后处理事件
        ngx_post_event(wev, &ngx_posted_events); //放入队列，暂时不处理
    } else { //不要求延后处理事件
        wev->handler(wev); //立即执行回调函数，处理事件
    }
} //写事件处理结束

```

启用负载均衡后的 `ngx_process_events()` 增加了对标志位 `NGX_POST_EVENTS` 的判断，不会调用 `ev->handler` 处理事件，也就没有了可能的阻塞，而是直接把事件加入延后处理队列 `ngx_posted_accept_events` 和 `ngx_posted_events`，前者专门存放 `accept` 事件，后者存放普通的读写事件。

处理事件

`ngx_process_events()` 执行之后，所有的事件都存放在了两个队列里，Nginx 优先处理 `ngx_posted_accept_events` 队列里的 `accept` 事件，然后立即释放锁：

```

// 位于 event/nginx_event.c
ngx_event_process_posted( //处理延后事件队列
    cycle, &ngx_posted_accept_events); //优先处理 accept 事件

if (ngx_accept_mutex_held) { //如果之前抢到了锁
    ngx_shmtx_unlock(&ngx_accept_mutex); //尽快释放锁供其他进程使用
}

```

对于 Linux `epoll` 来说，通常一次 `accept` 只接受一个连接，所以 `ngx_posted_accept_events` 队列里最多只会有相当于监听端口数量的事件，非常少，可以较快处理。当监听事件处理完就可以释放锁，让其他 `worker` 进程再去抢锁 `accept` 连接，随后可以“慢慢”处理定时器事件和存放在 `ngx_posted_events` 里的其他事件：

```

if (delta) { //如果消耗了一些时间，那么 delta>0
    ngx_event_expire_timers(); //查找超时事件并逐个处理
}

```

}

`ngx_event_process_posted(cycle, &ngx_posted_events);` //处理队列里的其他事件

流程图

Nginx 负载均衡的工作流程如图 14-10 所示。

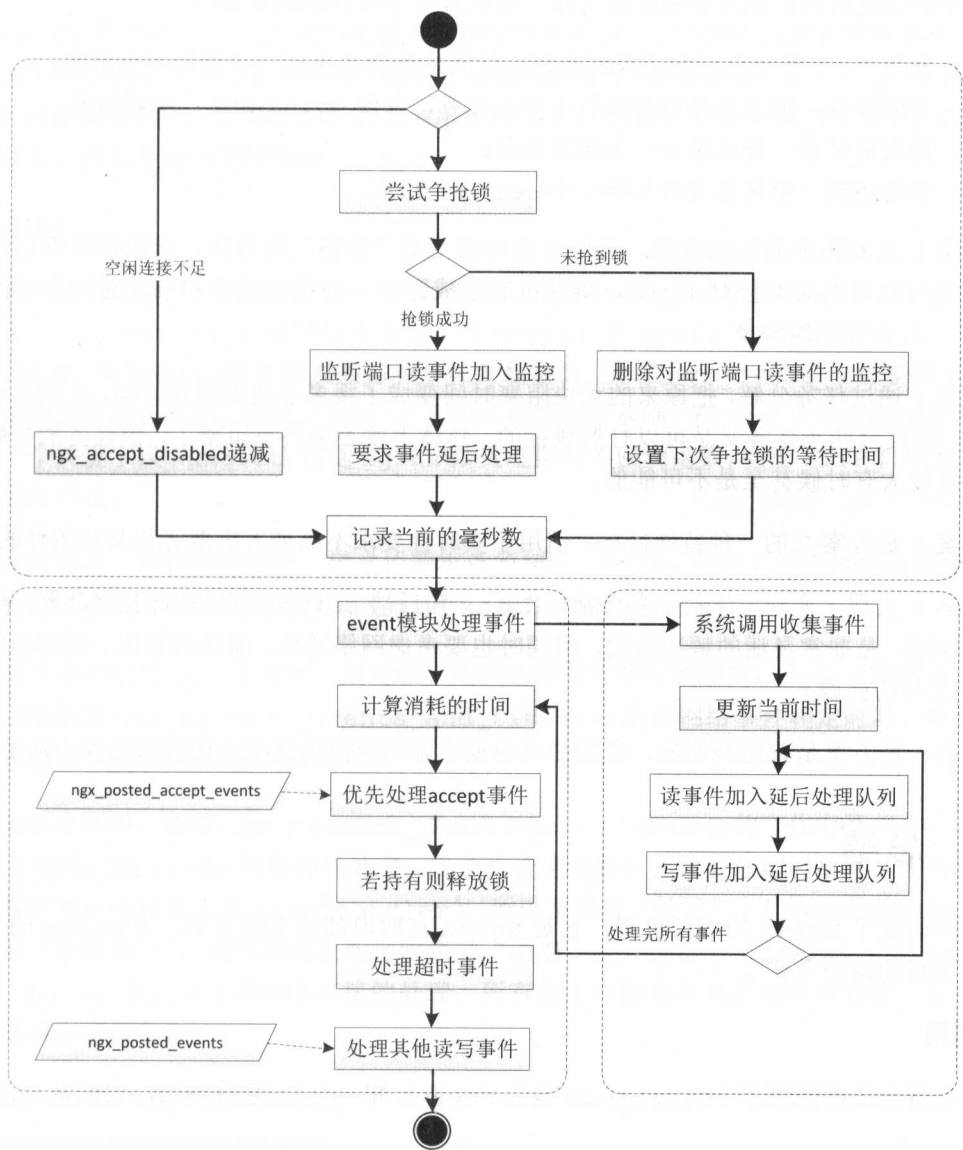


图 14-10 Nginx 负载均衡的工作流程

14.8.13 避免阻塞

Nginx 采用事件驱动机制，从处理事件的代码里（14.8.9 和 14.8.11 节）我们可以看到，Nginx 是“逐个”处理事件的，所以每个事件的 handler 函数不能有阻塞或者过度耗时的操作，否则不但会延误其他事件的处理，更会阻塞整个 Nginx 进程，造成服务能力下降。

对于那些耗时较长但又必须做的工作，可以有以下几种解决方案：

- 使用第 15 章里介绍的多线程机制，让其他线程去执行，主线程不会阻塞；
- 分解任务，把大任务拆分成若干个小任务，方便 Nginx 调度，缓解阻塞；
- 使用定时器，每次执行一小部分任务；
- 分离逻辑，把任务交给另外一个 server 处理。

方案 1 是最简单易行的方法，但同时也可能是最“偷懒”的方法。在线程里可以直接使用任意的阻塞调用而不会对 Nginx 事件机制造成影响，但执行效率和响应时间也可能不会太好。

方案 2 通过任务分解，把原来的大块阻塞时间变成了很多小块的阻塞时间，小到足够容忍的程度，而且这些小任务又是可以打散执行的，总体上就不会严重阻塞了，但任务的分解本身可能难度较大有时候甚至是不可能的。

方案 3 是方案 2 的一种特殊情况，适用于实时性要求不高或者需要等待某些条件的任务。

方案 4 实现了宏观层次的业务分离，Nginx 和后端 server 可以彼此独立部署，资源可以按需伸缩，是非常灵活的解决方案，但同时也要考虑网络通信、服务的发现、部署和监控等 Nginx 之外的问题。

没有一种方案是普适万能的，我们必须根据实际的应用场景综合比较来选择合适的方案。

14.9 总结

本章研究了 Nginx 的事件机制，它是 Nginx 宫殿内部的主动力量，为 Nginx 的运行提供源源不断的驱动力。

系统调用

我们首先简略回顾了 UNIX/Linux 里的 socket 和 epoll 系统调用，它们是 Nginx 事

件机制的基础。^①

socket 系列函数是目前网络编程事实上的标准,支持 TCP/UDP 等协议。为了提高效率,Nginx 使用的是非阻塞模式,调用方法与阻塞方式略有不同,关键的一点是每个函数调用时都不会阻塞等待,如果操作不能立即完成就会返回错误码 EAGAIN,需要之后结合 epoll 得到 ready 事件通知后再检查继续后面的操作。

epoll 是目前 Linux 系统里最强大的事件管理机制,由系统内核维护监控的事件列表,比传统的 select/poll 更加高效。epoll 也有两种工作模式,即 LT 和 ET,LT 是标准模式,而 ET 则是高速模式,必须搭配非阻塞调用才能工作。Nginx 对监听端口采用了 LT,其他的建立连接、收发数据等都用的是 ET 模式。

事件机制

Nginx 用 ngx_event_t、ngx_connection_t、ngx_listening_t、ngx_os_io_t、ngx_event_actions_t 等结构体实现了对 socket 和 epoll 系统调用的封装,分别代表事件、连接、监听端口、数据收发接口和事件处理接口,并在 ngx_cycle_t 里整合了连接池、事件池和监听端口数组,构成了 Nginx 事件机制的运行基础。在 ngx_listening_t 结构体里有一个关键字段 handler,它是 Nginx 接受连接时的回调函数,也是 HTTP 机制和 Stream 机制的入口点。

定时器是一类特殊的事件,用来检测事件的超时,Nginx 使用了红黑树结构来管理事件,把超时时间作为 key,从而可以高效地添加、删除和查找超时事件。

在启动初始化 epoll 模块、定时器、连接池和监听端口数组后,Nginx 就进入了无限循环,使用函数 ngx_process_events_and_timers() 处理网络事件和定时器,对外提供服务,它也是 Nginx 进程的核心函数。

如名字所示,函数 ngx_process_events_and_timers() 的工作有两部分,一是使用 epoll 收集已经 ready 的事件并处理,二是检查定时器红黑树处理超时事件。因为每个事件都关联有相应的回调函数(ev->handler),所以事件就“触发”了处理函数的执行,这就是所谓的“事件驱动”。通过周而复始地收集并“消费”accept/read/write/timedout 等事件,Nginx 就完成了对网络连接的处理,所有的工作都是在真正地处理数据,没有额外的阻塞等待和资源浪费。

Nginx 还内置负载均衡功能,可以在多个 worker 进程之间较均匀地分配新连接,它的

^① 这些系统调用同样可以用“man 2 xxx”查看 UNIX 参考手册获取更详细的信息。

实现要点是进程间的互斥锁和空闲连接阈值，只有空闲连接较多且抢锁成功的 worker 进程才能接受新连接。虽然负载均衡机制设计的很精妙，但它也增加了运行的成本，影响性能，所以现在 Nginx 不推荐使用此功能，读者需要留意。

了解了 Nginx 的事件机制的工作原理之后，我们就应当尽量避免开发过于耗时、可能导致进程阻塞的功能，从而更好地发挥 Nginx 的威力。

第 15 章

Nginx多线程机制

现代操作系统都支持线程（thread）概念，一个进程可以包含多个线程，每个线程共享进程里的所有系统资源。线程可以并发执行，简化异步操作，还能够改善性能。

线程带来的好处很多，但引发的问题也显而易见，最主要的就是由于并发导致的数据不同步，于是又出现了互斥量、条件变量、信号量等辅助机制，但这些机制又再次引发了更多的问题，例如使用不当造成的死锁，让多线程开发变成了一个复杂且颇具难度的课题。

Nginx 的核心运行机制是“多进程+事件驱动”，没有多线程的概念，但多线程毕竟是现实世界里客观存在的，所以在经过了长时间的沉默后，Nginx 终于从 1.7.11 开始正式引入了多线程机制。^①

多线程机制与事件处理机制在很多场景里都是彼此冲突的模式，但 Nginx 做了比较完善的设计，使两者能够和谐地配合工作，虽然算不上完美，但足以应付大多数需求。

要启用 Nginx 的多线程机制，必须在运行 configure 时使用选项“--with-threads”。

15.1 eventfd 系统调用

Linux 提供一个专有的系统调用 eventfd()，是实现 Nginx 事件通知机制的关键：

```
int eventfd(unsigned int initval, int flags); //Linux 专有系统调用
```

它创建一个特殊的“虚”文件描述符，由系统内核管理，专门用于用户态的事件通知，只

① 较早期的 Nginx 曾经有过多线程的试验性质代码，称为“old threads”，但很不完善，从来没有正式应用过，在新的多线程机制出现后它就从源码里彻底消失了。

要向它写入数据就可以如同 socket 一样被 epoll 监控到，也就是获取了“事件通知”。

15.2 pthread 系统调用

UNIX 系统下线程的标准是 POSIX 线程 (POSIX threads)，所以 UNIX 线程 API 也就被简称为 pthread。

pthread 标准包含非常多的接口，本节只简略介绍与线程创建和销毁相关的两个函数，其他的如 mutex、cond 等请读者自行参考 UNIX 手册。^①

15.2.1 pthread_create

创建线程需要使用函数 pthread_create()，声明是：

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine) (void *), void *arg);
```

函数创建一个新的线程，线程的 id 由第一个参数 thread 给出。

与创建进程的 fork() 类似，线程在创建后就会立即运行，但不是从主进程分支，而是执行第三个参数 start_routine 指定的函数，第四个参数 arg 则是函数指针 start_routine 的参数，所以，可以把线程看做是在进程的一个独立空间里运行的函数。

pthread_create() 的执行过程可以用图 15-1 来表示。

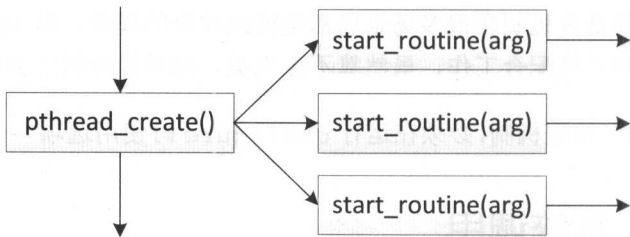


图 15-1 pthread_create() 的执行过程

pthread_create() 的入口参数比较“简陋”，如想要线程在运行时传递更多的参数，通常的做法是定义一个把参数“打包”的结构体，把结构体的指针作为 arg 参数。

^① pthread 系列函数位于 UNIX 参考手册的第 3 部分，属于库函数，并不是真正的系统调用（实际底层系统接口可能是 clone），但为了叙述方便本书仍然称它们为“系统调用”。

15.2.2 pthread_exit

线程运行的函数 `start_routine` 一旦退出（使用 `return`），线程也就结束了，但更好的方法是调用 `pthread_exit()`，它可以返回一些信息：

```
void pthread_exit(void *retval);           //结束线程
```

参数 `retval` 里的数据可以被其他线程使用 `pthread_join()` 获取，不过 Nginx 并没有这么用，只是个简单的 0（即 `NULL`）。

15.3 结构定义

线程自出现至今已有二十余年，关于多线程开发也已经有了很多成熟的模式，但这些模式大多比较复杂，不符合 Nginx 简单至上的设计理念。

在 Nginx 里只用到了两个多线程模式：生产者-消费者和线程池，本节介绍实现这两个模式所使用的数据结构。

15.3.1 ngx_thread_task_t

生产者-消费者（producer-consumer）模式是经典的多线程同步模式，两个线程操作一个共享的数据缓冲区，生产者线程产生数据，消费者线程处理数据。

生产者-消费者模式里处理的“数据”通常被称为“任务”（task），在 Nginx 里使用结构体 `ngx_thread_task_t` 来表示：

```
// 位于 core/nginx_core.h
typedef struct ngx_thread_task_s    ngx_thread_task_t;

// 位于 core/nginx_thread_pool.h
struct ngx_thread_task_s {           //需要线程处理的数据
    ngx_thread_task_t*    next;      //链表指针
    ngx_uint_t            id;        //任务的 id 号
    void*                  ctx;      //执行任务关联的额外数据
    void                  (*handler) ( //执行任务的函数指针
        void *data, ngx_log_t *log);
    ngx_event_t            event;    //任务关联的事件对象
};
```

Nginx 把多个等待线程运行的任务串成一个单向链表，成员 `next` 就是链表的指针。

成员 `id` 是任务的标识号，由一个全局变量计数器 `ngx_thread_pool_task_id` 产生，但它目前仅用于 Nginx 多线程机制的日志记录，任务执行时不使用。

成员 `handler` 是线程处理任务时需要执行的函数，它的参数就是 `ctx`，形式上与 `pthread_create()` 里的 `start_routine/arg` 有些类似。

最后一个成员 `event` 是 Nginx 多线程机制与事件机制两者之间的“桥梁”，它是一个“虚”事件对象，不在 Nginx 事件机制的事件池里，也不关联实际网络连接或者定时器，而只关联线程任务，所以里面的大部分字段都是无意义的，主要用到的是 `handler` 和 `data` 等，当线程完成任务时由事件机制回调。

所以在 `ngx_thread_task_t` 结构体里我们需要设置两个回调函数：一个是 `handler`，即在子线程里的运行逻辑；另一个是 `event.handler`，即子线程任务完成后主线程里的回调函数。

15.3.2 ngx_thread_pool_queue_t

Nginx 使用一个简单的队列来管理生产者-消费者模式里的数据缓冲区，它就是 `ngx_thread_pool_queue_t`：

```
// 位于 core/ngx_thread_pool.c
typedef struct {
    ngx_thread_task_t      *first;           // 队列首指针
    ngx_thread_task_t      **last;          // 队列尾指针
} ngx_thread_pool_queue_t;                 // 任务队列
```

队列里保存的元素是任务对象 `ngx_thread_task_t`，由 `next` 字段连接成单向链表。

注意它与 `ngx_queue_t` 的形式很相似，但两个成员表示的是队列的首尾指针，而不是侵入式节点的 `prev/next`，这种定义方式可以快速操作整个队列而不必逐个遍历节点。

15.3.3 ngx_thread_pool_t

线程池是多线程开发里的另一种常用模式，多用来配合生产者-消费者模式。它也是对象池模式的一个具体应用，通过预先创建出多个线程避免了反复创建销毁线程对象的成本，线程池里的每个线程都是“消费者”，在无限循环里处理数据。

Nginx 使用指令“`thread_pool`”配置线程池，格式是：

```
thread_pool name threads=number [max_queue=number];
```

指令配置了一个名为 `name` 的线程池，里面有 `threads` 个线程，任务队列最大长度是 `max_queue`，也就是最多可存放 `max_queue` 个等待运行的任务。

结构体 `ngx_thread_pool_t` 定义了 Nginx 使用的线程池：

```
// 位于 core/nginx_thread_pool.h
typedef struct ngx_thread_pool_s  ngx_thread_pool_t;

// 位于 core/nginx_thread_pool.c
struct ngx_thread_pool_s {
    ngx_thread_mutex_t      mtx;                //线程同步用的互斥量
    ngx_thread_pool_queue_t  queue;              //任务队列, 存放待线程运行的任务
    ngx_int_t               waiting;             //等待的任务数
    ngx_thread_cond_t        cond;               //线程同步用的条件变量

    ngx_log_t*              log;                 //线程机制里使用的日志对象

    ngx_str_t               name;                //线程池的名字
    ngx_uint_t              threads;             //线程的数量, 默认为 32 个线程
    ngx_int_t               max_queue;           //任务队列长度, 默认是 65535

    u_char*                 file;                //定义线程池的配置文件
    ngx_uint_t              line;                //定义线程池指令的行号
};
```

ngx_thread_pool_t 里的前四个成员是线程池的核心，queue 保存了待运行的任务，数量为 waiting 个，而 mtx 和 cond 用于多个线程对象同步操作队列。

其他的成员基本对应配置指令 “thread_pool”，含义都很明确。

15.3.4 结构关系图

Nginx 多线程机制里这些数据结构的关系如图 15-2 所示。

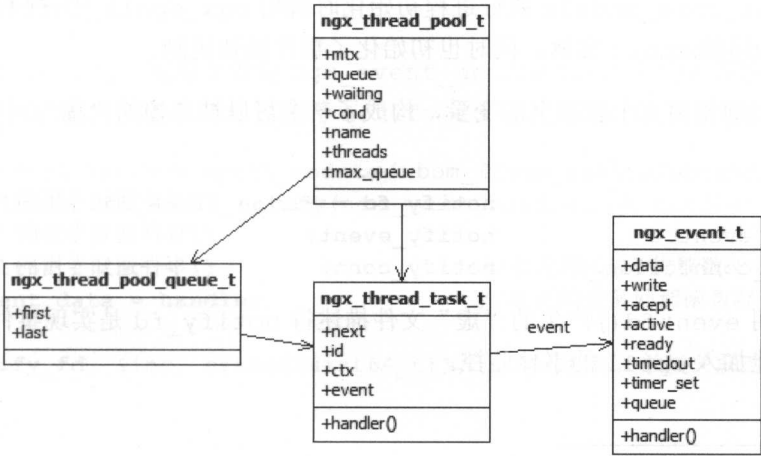


图 15-2 Nginx 多线程机制数据结构的关系

15.4 事件通知

在第 14 章研究事件机制时我们曾简单提到过事件通知机制，目前仅用于多线程，现在我们就来详细了解它的工作原理。

15.4.1 函数接口

事件通知是事件机制的一部分，也使用结构体 `ngx_event_actions_t` (14.4.6 节)，是里面的 `notify` 字段，定义是：

```
// 位于 event/nginx_event.h
typedef struct {
    ... //添加删除事件等函数
    ngx_int_t (*notify) (ngx_event_handler_pt handler); //事件通知函数
    ngx_int_t (*init) (ngx_cycle_t *cycle, ngx_msec_t timer);
    void (*done) (ngx_cycle_t *cycle);
} ngx_event_actions_t; //事件机制的访问接口
```

函数指针 `notify` 的含义是向事件机制发送一个通知，要求执行 `handler` 函数。^①

注意它的运行完全是在 Nginx 事件机制里，也就是说仅仅是在 `epoll` 里产生了一个特殊的“通知事件”（既不是网络事件也不是定时器事件），并不会立即执行 `handler`，只有当主线程的无限循环执行到 `ngx_process_events()` 时才能得到这个事件并执行 `handler`。

15.4.2 初始化

`ngx_event_core_module` 执行进程初始化时调用 `ngx_epoll_init()`，初始化每个 `worker` 进程专用的 `epoll` 实例，同时也初始化了事件通知机制。

事件通知机制使用三个静态全局变量，构成了一个可以被监控的“虚”对象：

```
// 位于 event/modules/nginx_epoll_module.c
static int notify_fd = -1; //事件通知专用的“虚”文件描述符
static ngx_event_t notify_event; //事件通知专用的“虚”事件对象
static ngx_connection_t notify_conn; //事件通知专用的“虚”连接对象
```

由系统调用 `eventfd()` 产生的“虚”文件描述符 `notify_fd` 是实现事件通知机制的关键，有了它才能加入 `epoll` 的事件监控。

① 早期版本的 Nginx 里没有 `notify`，这个位置上曾经是一个完全不同的 `process_change` 接口。

与“虚”文件描述符 `notify_fd` 对应的就是“虚”事件 `notify_event`，代表通知事件的发生和处理。因为 Nginx 的事件机制规定每个事件都必须关联一个连接对象，所以还需要有一个“虚”连接对象 `notify_conn`。

函数 `ngx_epoll_notify_init()` 初始化事件通知机制，代码摘要如下：

```
// 位于 event/modules/ngx_epoll_module.c
notify_fd = eventfd(0, 0); // 创建事件通知专用的虚文件描述符

notify_event.handler = // 事件通知的回调函数
    ngx_epoll_notify_handler;
notify_event.active = 1; // 事件活跃，即被 epoll 监控

notify_conn.fd = notify_fd; // 虚连接关联到虚文件描述符
notify_conn.read = &notify_event; // 虚连接关联到虚事件

ee.events = EPOLLIN|EPOLLET; // 监控读事件，ET 模式
ee.data.ptr = &notify_conn; // 保存连接对象，注意不需要失效标志位

epoll_ctl(ep, EPOLL_CTL_ADD, notify_fd, &ee); // 加入 epoll 监控
```

这段代码比较简单，首先用 `eventfd()` 产生“虚”文件描述符，然后设置“虚”事件的回调函数 `ngx_epoll_notify_handler`，再把这两个对象关联到“虚”连接对象。由于事件通知是单向的，所以只需要一个读事件就够了。最后调用 `epoll_ctl()` 把它加入 `epoll` 的监控，这样如果有线程向 `notify_fd` 写入了数据，那么就会触发可读事件，`epoll_wait()` 就能够收到事件通知，进而执行 `ngx_epoll_notify_handler`。

15.4.3 发送通知

函数宏 `ngx_notify`（实际上就是 `ngx_event_actions.notify`）实现事件通知，它的工作非常简单，就是向“虚”文件描述符 `notify_fd` 写入数据，触发关联的读事件：

```
// 位于 event/modules/ngx_epoll_module.c
static ngx_int_t ngx_epoll_notify(ngx_event_handler_pt handler)
{
    static uint64_t inc = 1; // 永远写入 64 位整数 1
    notify_event.data = handler; // 真正的业务回调函数存储在 data 里

    write(notify_fd, &inc, sizeof(uint64_t)); // 写入数据，触发关联的读事件

    return NGX_OK;
}
```

因为 `notify_event` 的 `handler` 已经被固定为 `ngx_epoll_notify_handler`, 所以代码里用 `notify_event` 的 `data` 字段存储了真正要执行的回调函数, 由 `ngx_epoll_notify_handler` 间接运行, 这是函数的关键操作。

向“虚”文件描述符里写入的数据没有实际的意义, Nginx 仅用了一个 64 位的整数, 固定为值 1。

15.4.4 处理通知

`ngx_notify` 执行后, `epoll` 就会监控到 `notify_fd` 上发生的可读事件, 在 `ngx_epoll_process_events()` 里调用关联的 `notify_event.handler` 也就是 `ngx_epoll_notify_handler` 处理, 从 `ev->data` 里获取之前 `ngx_notify` 存放的真正回调函数并执行, 代码摘要如下:

```
// 位于 event/modules/ngx_epoll_module.c
static void ngx_epoll_notify_handler(ngx_event_t *ev)
{
    if (++ev->index == NGX_MAX_UINT32_VALUE) {           //触发错误校验
        ev->index = 0;                                     //清空 index 计数器

        n = read(notify_fd, &count, sizeof(uint64_t)); //读出写入的数据

        if ((size_t) n != sizeof(uint64_t)) {             //只检查 n 的位数
            ngx_log_error(...);                             //不是 64 位则记录错误日志
        }
    }                                                       //错误校验结束

    handler = ev->data;                                     //取出真正的业务回调函数
    handler(ev);                                           //执行真正的业务回调函数
}
```

因为“虚”文件描述符 `notify_fd` 里不是实际的数据 (永远是 1), 对它的操作绝大多数情况下是不会出错的, 不应该做频繁的错误检查。所以 Nginx 使用了 `ngx_event_t` 里“无用”的字段 `index` 作为错误检查的计数器, 每触发一次事件通知就把它加 1, 只有当它增加到 `NGX_MAX_UINT32_VALUE` (`0xffffffff`) 时才进行错误检查, 这样就大大降低了错误检查的频率。^①

① `ev->index` 仅是在 `epoll` 机制里无用, 其他的事件机制如 `select`、`poll`、`kqueue` 都会用到。

15.5 运行机制

`ngx_thread_pool_module` 是 Nginx 多线程机制的实现模块，它属于 `core` 模块，集中管理线程池，并实现多线程相关的运行逻辑。

15.5.1 完成任务队列

Nginx 使用一个全局变量 `ngx_thread_pool_done` 保存所有执行完毕的任务：

```
// 位于 core/nginx_thread_pool.c
static ngx_thread_pool_queue_t  ngx_thread_pool_done; //完成任务的队列
```

在多线程环境下对链表的操作是不安全的，必须要使用锁，所以还有一个配合它的自旋锁：

```
// 位于 core/nginx_thread_pool.c
static ngx_atomic_t      ngx_thread_pool_done_lock; //队列的保护锁
```

15.5.2 创建线程池

`ngx_thread_pool_module` 的配置结构体定义是：

```
// 位于 core/nginx_thread_pool.c
typedef struct {
    ngx_array_t      pools; //所有的线程池定义
} ngx_thread_pool_conf_t; //线程池配置结构体
```

它的结构很简单，成员 `pools` 保存了所有的 “`thread_pool`” 指令配置的线程池，元素的类型是 `ngx_thread_pool_t`。

进程初始化时会执行模块的 `ngx_thread_pool_init_worker()` 函数，在这里遍历线程池数组，创建线程池：

```
// 位于 core/nginx_thread_pool.c
tpp = tcf->pools.elts; //线程池数组
for (i = 0; i < tcf->pools.nelts; i++) { //遍历线程池数组
    ngx_thread_pool_init(tpp[i], ...); //创建线程池
}
```

函数 `ngx_thread_pool_init()` 执行具体的创建线程池工作，代码摘要如下：

```
// 位于 core/nginx_thread_pool.c
if (ngx_notify == NULL) { //要求必须有事件通知机制
    return NGX_ERROR; //否则多线程无法工作
```

```

}

ngx_thread_pool_queue_init(&tp->queue);           //初始化线程池任务队列

ngx_thread_mutex_create(&tp->mtx, log);           //系统调用创建互斥量
ngx_thread_cond_create(&tp->cond, log);           //系统调用创建条件变量

for (n = 0; n < tp->threads; n++) {               //根据配置的线程数量, 创建线程
    err = pthread_create(                          //系统调用创建线程
        &tid, &attr, ngx_thread_pool_cycle, tp); //传递线程执行的函数
}
}

```

这样, 一个线程池就完成了初始化。最开始任务队列是空的, 启动了 n 个线程, 每个线程运行的是函数 `ngx_thread_pool_cycle()`, 使用条件变量阻塞等待任务队列 (见 15.5.5 节)。

创建好的线程池可以用函数 `ngx_thread_pool_get()` 获取:

```
ngx_thread_pool_t *ngx_thread_pool_get(ngx_cycle_t *cycle, ngx_str_t *name);
```

15.5.3 创建任务

通常情况下 Nginx 主线程 (事件机制) 是任务的生产者, 函数 `ngx_thread_task_alloc()` 创建一个任务对象:

```

// 位于 core/nginx_thread_pool.c
ngx_thread_task_t* ngx_thread_task_alloc(ngx_pool_t *pool, size_t size)
{
    task = ngx_palloc(pool,                      //内存池分配内存
        sizeof(ngx_thread_task_t) + size);       //多分配了 size 个字节
    task->ctx = task + 1;                         //设置 ctx 数据指针

    return task;                                  //返回创建好的任务对象
}

```

函数的代码虽然不多, 但用法却值得注意。

首先要看到的是它使用了内存池, 可以是 `cycle`、`connection` 或者是 `request` 对象里的内存池, 因为 Nginx 内存池不是线程安全的, 所以 `ngx_thread_task_alloc()` 最好是在主线程里调用, 否则可能会导致内存池数据结构被破坏。

其次要注意函数的第二个参数 `size`, 它表示的是执行任务关联的数据 `ctx` 的大小

(15.3.1 节), 在创建任务对象时额外分配了 `size` 个字节, 相当于把 `ctx` 数据与 `ngx_thread_task_t` 做了一个“强制绑定”, `ctx` 指针直接指向结构体后的位置即可。^①

`ngx_thread_task_t` 里存储的数据非常重要, 线程的运行和完成后的回调都依赖于它, `ctx` 里存放的是线程运行使用的数据, 而 `event.data` 则可以用来存储其他数据。一种常用的做法是 `ctx` 结构里额外存储连接、请求等信息, `event.data` 存储 `ngx_thread_task_t` 指针, 这样事件机制回调时就可以通过这个指针同时得到任务和请求的数据。

15.5.4 投递任务

有了任务对象, 就可以调用 `ngx_thread_task_post()` 投递到某个线程池里, 让线程运行, 函数的代码摘要如下:

```
// 位于 core/nginx_thread_pool.c
ngx_int_t
ngx_thread_task_post(ngx_thread_pool_t *tp, ngx_thread_task_t *task)
{
    if (task->event.active) {                //active 表示任务已经放入任务队列
        return NGX_ERROR;                    //不重复投递任务
    }

    if (ngx_thread_mutex_lock(&tp->mtx) != NGX_OK) { //互斥量锁定保护任务队列
        return NGX_ERROR;
    }

    if (tp->waiting >= tp->max_queue) {        //检查等待运行的任务数量
        ngx_thread_mutex_unlock(&tp->mtx);    //超过上限则不投递, 解锁互斥量
        return NGX_ERROR;
    }

    task->event.active = 1;                    //active 表示任务已经放入任务队列

    task->id = ngx_thread_pool_task_id++;      //设置任务的 id 号
    task->next = NULL;                         //后续指针置空, 即队列末尾

    if (ngx_thread_cond_signal(&tp->cond) != NGX_OK) { //条件变量通知
        (void) ngx_thread_mutex_unlock(&tp->mtx);
        return NGX_ERROR;
    }
}
```

① 这其实是 C 语言开发时一种常用的 hack 技巧, 原因在于 C 语言可以使用强制转型对指针 (即内存地址) 任意解释。


```

*tp->queue.last = task;           //把任务加入队列的末尾
tp->queue.last = &task->next;      //队列尾指针调整
tp->waiting++;                     //等待运行的任务数量增加

(void) ngx_thread_mutex_unlock(&tp->mtx); //投递完成，解锁互斥量

return NGX_OK;
}

```

Nginx 使用互斥量 `mtx` 来保护 `waiting`、`queue`、`ngx_thread_pool_task_id` 等线程间共享的变量，在锁住 `mtx` 后就可以检查队列长度，如果队列未满就把任务加到末尾。

条件变量 `cond` 实现了生产者线程向消费者线程的通知，这样在线程池里某个阻塞等待的线程就会被唤醒，检查任务队列去运行任务。

15.5.5 执行任务

函数 `ngx_thread_pool_cycle()` 是线程运行的函数，它的名字与进程机制的 `ngx_worker_process_cycle` 很相似，里面同样是一个无限循环，只不过它处理的不是事件，而是任务。

函数 `ngx_thread_pool_cycle()` 的代码摘要如下：

```

// 位于 core/nginx_thread_pool.c
static void * ngx_thread_pool_cycle(void *data)
{
    ngx_thread_pool_t *tp = data;           //参数转型为线程池对象

    for ( ;; ) {                             //无限循环执行任务
        ngx_thread_mutex_lock(&tp->mtx, tp->log); //锁定互斥量

        tp->waiting--;                         //等待任务数减少

        while (tp->queue.first == NULL) {      //任务队列空则阻塞等待
            (ngx_thread_cond_wait(&tp->cond, &tp->mtx, tp->log);
            //直至收到条件变量的通知

        task = tp->queue.first;                //取队列里的第一个任务
        tp->queue.first = task->next;           //调整队列头指针

        if (tp->queue.first == NULL) {         //如果队列空
            tp->queue.last = &tp->queue.first; //调整队列尾指针
        }

        ngx_thread_mutex_unlock(&tp->mtx, tp->log); //取任务完毕，解锁互斥量
    }
}

```

```

task->handler(task->ctx, tp->log);           //执行任务!
task->next = NULL;                          //任务后继指针置空

ngx_spinlock(&ngx_thread_pool_done_lock,...); //自旋锁保护完成任务队列

*ngx_thread_pool_done.last = task;          //任务加到完成队列末尾
ngx_thread_pool_done.last = &task->next;    //调整队列尾指针

ngx_memory_barrier();                      //内存顺序一致性保护
ngx_unlock(&ngx_thread_pool_done_lock);     //解锁自旋锁

ngx_notify(ngx_thread_pool_handler);       //发出事件通知, 完成任务回调
}
}

```

`ngx_thread_pool_cycle()` 所在的线程是消费者, 它使用条件变量等待任务队列, 如果队列里没有任务它就会持续阻塞。一旦主线程使用 `ngx_thread_task_post()` 把任务投递进了线程池里的队列那么线程就会被唤醒, 取出队列里的一个任务, 然后立即释放锁以便其他线程获取任务 (与 Nginx 负载均衡机制里的多进程争抢锁比较类似)。

拿到任务对象后, Nginx 就在线程里执行 `task->handler`, 它可以是任意的功能, 阻塞或非阻塞都允许, 例如读写磁盘、复杂计算、远程过程调用等, 但必须要遵守多线程编程的准则, 访问主线程里的数据时需要小心谨慎, 不能破坏进程机制、事件机制相关的数据结构。

任务执行完毕后, Nginx 使用自旋锁保护完成队列, 把它挂到 `done` 队列的末尾, 最后调用 `ngx_notify` (15.4.3 节), 通知主线程里的事件机制执行回调函数 `ngx_thread_pool_handler`。注意 `done` 队列只有一个, 所有线程池里完成的任务都存放在这一个队列里。

`ngx_thread_pool_cycle()` 的工作流程如图 15-3 所示。

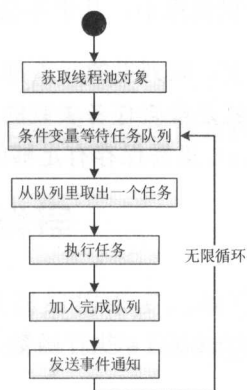


图 15-3 `ngx_thread_pool_cycle()` 的工作流程

15.5.6 任务完成回调

线程池里的线程是并发运行的，每完成一个任务都会发送一个事件通知，当主线程运行到 `ngx_epoll_process_events()` 时就会收到通知，执行指定的回调函数。

线程池的回调函数 `ngx_thread_pool_handler()` 运行在主线程里，它检查任务完成队列，逐个执行任务关联的事件对象，代码摘要如下：

```
// 位于 core/nginx_thread_pool.c
ngx_spinlock(&ngx_thread_pool_done_lock, ...); //自旋锁保护完成任务队列

task = ngx_thread_pool_done.first;           //取整个队列
ngx_thread_pool_done.first = NULL;           //队列头指针置空
ngx_thread_pool_done.last = &ngx_thread_pool_done.first; //队列尾指针置空

ngx_memory_barrier();                        //内存顺序一致性保护
ngx_unlock(&ngx_thread_pool_done_lock);      //解锁自旋锁

while (task) {                               //遍历完成任务链表
    event = &task->event;                     //取任务关联的事件
    task = task->next;                         //指针后移

    event->complete = 1;                      //事件已完成
    event->active = 0;                        //事件不活跃，即不在线程里运行

    event->handler(event);                    //执行事件的回调函数
}
```

这段代码里对任务队列的操作手法很值得学习。由于队列是个链表，所以 Nginx 直接用头节点获取了整个链表，随后就调整队列的指针把队列“清空”，整个过程只操作了三个指针，没有数据的拷贝，因而自旋锁的代价也就很小，几乎不会产生阻塞。

自旋锁解除之后，任务队列被清空，队列里的元素都保存在了 `task` 指向的链表里。由于线程任务已经完成，所以要做的工作就是检查任务关联的事件，执行事件的回调函数，重新回到投递任务之前的流程（`task->event` 里需保存有足够的场景恢复信息），有些类似 14.4.7 节对延后事件队列的处理。

15.5.7 销毁线程池

进程结束时 `ngx_thread_pool_module` 执行函数 `ngx_thread_pool_destroy()`，清空任务队列，销毁线程池：

```
// 位于 core/nginx_thread_pool.c
```

```

static void ngx_thread_pool_destroy(ngx_thread_pool_t *tp)
{
    ngx_thread_task_t    task;                //要求线程结束的任务
    volatile ngx_uint_t  lock;                //线程结束的标志量

    task.handler = ngx_thread_pool_exit_handler; //特殊的结束任务
    task.ctx = (void *) &lock;                //结束线程使用的数据

    for (n = 0; n < tp->threads; n++) {        //为每一个线程投递一个结束任务
        lock = 1;                             //设置标志量
        ngx_thread_task_post(tp, &task);        //投递一个结束任务

        while (lock) {                         //检查线程是否执行了任务
            ngx_sched_yield();                 //避免占用 CPU, 让出主线程执行权
        }

        task.event.active = 0;                 //重用任务, 再投递进队列
    }

    ngx_thread_cond_destroy(&tp->cond, tp->log); //销毁条件变量
    ngx_thread_mutex_destroy(&tp->mtx, tp->log); //销毁互斥量
}

```

在销毁线程池时 Nginx 向任务队列里投递了一个特殊的结束任务，任务的执行函数是 `ngx_thread_pool_exit_handler`，非常简单，只是修改标志量，然后调用 `pthread_exit(0)` 结束线程：

```

// 位于 core/ngx_thread_pool.c
static void ngx_thread_pool_exit_handler(void *data, ngx_log_t *log)
{
    ngx_uint_t *lock = data;                //获取传递的整数变量
    *lock = 0;                             //改为 0 表示线程已经执行了
    pthread_exit(0);                         //结束线程
}

```

在连续投递多次结束任务后，线程池里的线程都会收到任务并执行，最终结束所有的线程。

15.6 在模块里使用多线程

Nginx 多线程机制用到的数据结构不多，故可以很容易地封装为 C++ 类，例如 `NgxThreadTask<T>`、`NgxThreadPool`，为了节约篇幅不再列出它们的代码，请读者参考 GitHub 资源。

作为示例，本章将使用 C++ 开发一个简单的使用多线程的 Nginx 模块。

15.6.1 模块设计

使用多线程的模块设计如下：

- 模块名是 `ndg_thread_module`，是内容处理模块；
- 模块使用 `content handler` 的方式注册处理函数，发送一个字符串；
- 配置指令是 `ndg_thread_hello`，确定使用的线程池名字。

15.6.2 配置信息类

`NdgThreadConf` 定义模块的配置信息，只有一个 `ngx_str_t` 成员，存储模块使用的线程池名字：

```
class NdgThreadConf final
{
public:
    typedef NdgThreadConf      this_type;           //简化类型定义
public:
    ngx_str_t                  pool_name;           //使用的线程池名字
public:
    static void* create(ngx_conf_t* cf)              //创建配置信息类
    { return NgxPool(cf).alloc<this_type>(); }

    static this_type& cast(void* conf)               //方便的转型函数
    { return *reinterpret_cast<this_type*>(conf); }
};
```

15.6.3 业务逻辑类

与使用子请求类似，我们用类 `NdgHelloTask` 来实现线程相关的主要逻辑，类 `NdgThreadHandler` 向线程池投递这个任务。

任务的 ctx

开发 Nginx 的多线程功能首先要做的就是定义任务的 ctx，存储在线程里运行的一切必要信息。我们的模块比较简单，只需要一块用于输出字符串的内存：

```
class NdgHelloTaskCtx final
{
public:
    typedef NdgHelloTaskCtx this_type;           //简化类型定义
```

```

public:
    ngx_buf_t*          buf = nullptr;           //线程使用的内存空间
    ngx_thread_task_t*   task = nullptr;         //任务对象指针
    ngx_http_request_t*  r = nullptr;           //请求对象指针
public:
    static this_type* cast(void* p)              //方便的转型函数
    { return reinterpret_cast<this_type*>(p); }
};

```

为了方便完成任务后的回调，NdgHelloTaskCtx 里面还保存了任务对象和请求对象，所以使用这个 ctx 就可以随时得到线程任务和 HTTP 请求的所有信息。

任务对象

类 NdgHelloTask 实现了模块的主要业务逻辑，定义如下：

```

class NdgHelloTask final                                //一个具体的线程任务
{
public:
    typedef NdgHelloTaskCtx          ctx_type;         //简化类型定义
    typedef NgxThreadTask<ctx_type> task_type;         //封装 Nginx 线程任务

    typedef NdgHelloTask              this_type;
public:
    ...                                                //其他成员函数，见后
};

```

注意在代码里我们使用了类 NgxThreadTask，它封装了 ngx_thread_task_t，模板参数就是任务 ctx 的类型。

创建任务

NdgHelloTask 的静态成员函数 init() 创建 ngx_thread_task_t 结构体，并设置里面的各个成员，为投递任务做准备：

```

public:
    static ngx_thread_task_t* init(ngx_http_request_t* r) //创建任务对象
    {
        task_type task = task_type::create(r->pool);    //创建结构体

        auto ctx = task.ctx();                          //获取 ctx，自动转型

        ctx->buf = NgxPool(r->pool).buffer(100);         //分配内存
        ctx->r = r;                                       //保存请求对象指针
        ctx->task = task.get();                          //保存任务对象指针
    }

```

```

    task.handler(&this_type::task_handler);           //设置任务的执行函数

    task.event().data(task.get());                   //事件保存任务对象指针
    task.event().handler(&this_type::event_handler); //设置事件的回调函数

    return task;                                     //返回创建好的对象
}

```

函数设置了任务运行必需的数据，最重要的是 ctx 里的内存 buf，必须在主线程里事先分配好，之后是设置两个函数，分别是在子线程里运行的 task_handler 和在主线程里运行的 event_handler。

运行任务

模块在线程里执行的任务很简单，只是输出一个格式化的字符串，当然也可以改为其他任何操作，例如读写磁盘、访问数据库等：

```

public:
    static void task_handler(void *data, ngx_log_t *log)    //运行任务
    {
        auto ctx = ctx_type::cast(data);                  //转型为正确的 ctx 对象

        ngx_buf_t buf = ctx->buf;                          //获取之前分配的内存空间

        ngx_str_t s = buf.boundary();                     //转换成字符串方便操作
        ngx_string(s).printf("%s", "hello nginx thread\n"); //格式化字符串
        buf.produce(s.len);                                //缓冲区长度增加

        ngx_log_error(log).print(                          //使用 task->id 记录日志
            "task %d run in thread", ctx->task->id);
    }

```

任务完成回调

当任务在线程池里运行完后会发出事件通知，回调之前在事件对象里设置的函数，这时我们需要恢复之前被中断的请求，取出线程任务返回的数据，发给客户端：

```

public:
    static void event_handler(ngx_event_t *ev)            //任务完成回调
    {
        task_type task =                                 //获取任务对象
            reinterpret_cast<ngx_thread_task_t*>(ev->data);
        auto ctx = task.ctx();                           //获取任务的 ctx 数据
    }

```

```

auto r = ctx->r; //获取之前保存的请求对象

--r->main->blocked; //阻塞数减少
--r->main->count; //引用计数减少

NgxBuf buf = ctx->buf; //获得线程的运行结果

NgxResponse resp(r); //响应对象

resp.length(buf.size()); //设置响应体长度
resp.status(NGX_HTTP_OK); //设置状态码
resp.send(buf); //发送响应数据
resp.eof(); //发送响应结束标志
}

```

在请求里使用线程任务时不仅要用到 `ngx_http_request_t` 里的引用计数 `count`，还要用到 `blocked`，后者专门用来标记当前请求里正在“阻塞”运行的任务数量，两个字段都需要在投递任务时加 1，在完成任务时减 1，否则 HTTP 框架会无法正确处理请求。

投递任务

类 `NdgThreadHandler` 处理 HTTP 请求，它的工作就是调用 `NdgHelloTask` 生成任务，然后把任务投递到线程池运行，返回 `NGX_DONE`，后续的处理都由 `NdgHelloTask` 执行：

```

class NdgThreadHandler final //处理 HTTP 请求
{
public:
    typedef NdgThreadHandler this_type; //简化类型定义
    typedef NdgThreadModule this_module;
    typedef NdgHelloTask hello_task; //使用的任务
public:
    static ngx_int_t handler(ngx_http_request_t *r) //处理 HTTP 请求
    {
        NgxRequest req(r); //HTTP 请求对象
        req.body().discard(); //丢弃请求体

        auto task = hello_task::init(r); //创建一个任务

        auto& lcf = this_module::conf().loc(r); //获取配置信息

        NgxThreadPool tp(lcf.pool_name); //获取线程池
        tp.post(task); //向线程池投递任务

        ++r->main->blocked; //阻塞数增加
        ++r->main->count; //引用计数增加
    }
}

```



```

    return NGX_DONE;           //请求暂未处理完，需要等待任务完成事件才能继续处理
}
};

```

15.6.4 测试验证

这里不再列出 `ndg_thread_module` 的模块集成和编译部分的实现代码（都很简单），直接进入测试验证环节。

在 Nginx 的配置文件里首先要定义一个线程池：

```

thread_pool hello_pool threads=2;           #线程池里有两个线程

```

然后是使用多线程的 location：

```

location /thread {                       #使用多线程的 location
    ndg_thread_hello hello_pool;        #指定使用的线程池
}

```

使用 curl 访问 /thread，就会得到一个字符串：

```

curl http://localhost/thread           #执行 curl 命令
hello nginx thread                     #模块的输出结果

```

查看 Nginx 的运行日志可以验证任务确实是在线程里执行了：

```

2017/xx/xx 15:00:48 [error] 28081#28082: task 0 run in thread

```

15.7 总结

本章我们研究了 Nginx 的多线程机制，它是 Nginx 宫殿的副动力室，为 http、stream 等模块提供额外的运行动力。

Nginx 多线程机制基于 Linux 的专有系统调用 `eventfd()`，它创建一个特殊的“虚”文件描述符，可以被 `epoll` 监控，没有它就无法实现多线程与事件机制的结合。

关于多线程开发有很多成熟的模式，但在 Nginx 里只用到了生产者-消费者和线程池。Nginx 主线程里的事件机制扮演了生产者的角色，线程池里的线程则是消费者，结构体 `ngx_thread_task_t`、`ngx_thread_pool_queue_t` 和 `ngx_thread_pool_t` 分别是任务、任务队列和线程池。

在主线程里，我们要填充任务对象 `ngx_thread_task_t` 里的各个字段，关键是设置子

线程里的运行函数 `handler` 和主线程里的回调函数 `event.handler`，还要把所需的数据存储在 `ctx` 里。

有了任务对象，就可以调用 `ngx_thread_task_post()` 把任务投递到线程池，让线程去运行任务，而主线程不会被任务阻塞。注意投递后需要操作 `ngx_http_request_t` 里的字段 `count` 和 `blocked`，增加引用计数，否则任务完成后 HTTP 框架会无法正确处理请求。

线程池里的每一个线程都在执行函数 `ngx_thread_pool_cycle()`，里面也是个无限循环，使用条件变量阻塞等待任务队列。一旦有任务线程就会被唤醒，从队列里取出任务，然后执行任务的 `handler`，执行完毕后放入线程池的 `done` 队列，调用 `eventfd()` 向主线程发送事件通知。

在主线程的事件机制里，收到事件通知就会执行函数 `ngx_thread_pool_handler()`，检查 `done` 队列，逐个执行任务的 `event.handler`，之前被中断的请求将从这里继续处理，可以从任务的 `ctx` 里取出线程计算的结果。

就这样，主线程与线程池各自独立运行，通过任务队列和完成队列相互交换数据：主线程产生任务，投递到任务队列里；线程池里的多个线程不停地处理队列里的任务，执行完毕的任务放入完成队列；主线程再从完成队列里获取线程池的处理结果。

本章的最后我们使用 C++ 实现了一个简单的多线程模块，演示了 Nginx 多线程开发的基本流程，读者可以参考它实现更复杂的功能。

第 16 章

Nginx Stream机制

Nginx 从 1.9.0 开始引入了 stream 模块,可以直接处理 TCP/UDP 协议,扩展了 Nginx 的能力范围,不再限于 HTTP/Mail Server,而变成了一个更通用的 Server 软件。

经过了两年多的发展,目前 Nginx 的 Stream 机制已经初步确立,虽然没有像 HTTP 机制那样成熟繁多的功能模块,但基本上也很齐全了,支持限速、访问控制、反向代理、SSL 等常用的功能,灵活组合现有的这些模块就可以搭建出一个较完善的 TCP/UDP 服务器。

当然,想要更充分地利用 Nginx 的 Stream 机制就必须了解它的工作原理,然后在它的框架里进行二次开发,本章就将讲解这方面的知识(由于涉及的内容较多,阅读时最好结合第 6 章的模块体系、第 13 章的进程机制和第 14 章的事件机制)。

目前 Nginx 默认不启用 Stream 机制,在编译时必须使用选项 “--with-stream”。

16.1 模块体系

Nginx 使用 stream 模块处理 TCP/UDP 协议,它是一种独立类型的模块,类型标识是 `NGX_STREAM_MODULE` (“STRM”)。

Nginx 的 Stream 框架与 HTTP 框架类似,不过因为 TCP/UDP 协议是原始字节流,不像 HTTP 那么复杂,所以内部实现细节上要简单一些,较容易理解,两者可以互相参考对比学习。

16.1.1 函数指针表

stream 模块使用的函数指针表是 `ngx_stream_module_t`,用于“子类化”模块结构体 `ngx_module_t` 里的 `ctx` 字段,定义如下:

```
// 位于 stream/nginx_stream.h
typedef struct {
    ngx_int_t    (*preconfiguration) (ngx_conf_t *cf);           //配置解析前
    ngx_int_t    (*postconfiguration) (ngx_conf_t *cf);          //配置解析后

    void*        (*create_main_conf) (ngx_conf_t *cf);           //创建 main 配置
    char*        (*init_main_conf) (ngx_conf_t *cf, void *conf); //初始化 main 配置

    void*        (*create_srv_conf) (ngx_conf_t *cf);           //创建 srv 配置
    char*        (*merge_srv_conf) (ngx_conf_t *cf, void *prev, //初始化 srv 配置
                                    void *conf);

} ngx_stream_module_t;
```

ngx_stream_module_t 的定义与 ngx_http_module_t 很像，都是配置解析相关的回调函数，调用的时机和作用也相同，但因为 TCP/UDP 协议没有“URI”、“location”的概念，所以不需要 create_loc_conf/merge_loc_conf 这两个函数。

同样因为这个原因，在 Nginx 里 stream{} 块里只有 server{} 层次，server{} 也不能嵌套，结构上简单了很多。

每一个 stream 模块都必须定义自己的 ngx_stream_module_t，实现模块在 main 域和 srv 域的配置初始化和合并工作。

16.1.2 基础模块

模块 ngx_stream_module 属于 core 模块，组织所有的 stream 模块，在 ngx_cycle->conf_ctx 数组里创建存储配置结构体的内存空间，功能与 ngx_events_module 差不多。

ngx_stream_module 使用结构体 ngx_stream_conf_ctx_t 来存储 stream 模块的配置数据，定义是：

```
// 位于 stream/nginx_stream.h
typedef struct {
    void**      main_conf;           //main 域的存储数组
    void**      srv_conf;            //srv 域的存储数组
} ngx_stream_conf_ctx_t;
```

ngx_stream_conf_ctx_t 与 http 模块里的 ngx_http_conf_ctx_t 类似，保存 stream 模块的两个层次的配置数据，main_conf 存储 stream{} 级别的配置信息，srv_conf 存储 server{} 级别的配置信息，这两个字段都是一维数组，里面的元素是 void*，由具体 stream 模块的 create_xxx_conf 函数创建。

ngx_stream_module 使用配置指令“stream”解析 stream{...} 配置块，分配存储

空间，配置所有的 stream 模块，最后初始化处理引擎、变量数组和监听端口，完成 stream 框架的初始化工作。

这里我们暂时只关注模块的配置部分，函数 `ngx_stream_block()` 代码摘要如下：

```
// 位于 stream/ngx_stream.c
if (*(ngx_stream_conf_ctx_t **) conf) {    //检查是否重复定义了 stream{}
    return "is duplicate";                //只能有一个 stream{}配置块
}

ctx = ngx_palloc(                          //创建配置结构体，里面是两个数组
    cf->pool, sizeof(ngx_stream_conf_ctx_t));

*(ngx_stream_conf_ctx_t **) conf = ctx;    //存放到 cycle->conf_ctx 里

ngx_stream_max_module =                   //计算 stream 模块的数量，初始化 ctx_index
    ngx_count_modules(cf->cycle, NGX_STREAM_MODULE);

ctx->main_conf = ngx_palloc(...);          //创建配置数组，用于存储 main 域配置
ctx->srv_conf = ngx_palloc(...);           //创建配置数组，用于存储 srv 域配置

for (m = 0; cf->cycle->modules[m]; m++) {  //遍历模块数组，只处理 stream 模块
    module = cf->cycle->modules[m]->ctx;    //获取模块的 ctx 函数表
    mi = cf->cycle->modules[m]->ctx_index;  //获取模块的 ctx_index 索引

    ctx->main_conf[mi] = module->create_main_conf(cf); //创建模块的 main 配置
    ctx->srv_conf[mi] = module->create_srv_conf(cf);  //创建模块的 srv 配置
}

pcf = *cf;                                //暂存之前配置解析的 ctx
cf->ctx = ctx;                             //设置本配置块的 ctx，即配置数组

for (m = 0; cf->cycle->modules[m]; m++) {  //遍历模块数组，只处理 stream 模块
    module = cf->cycle->modules[m]->ctx;    //获取模块的 ctx 函数表
    module->preconfiguration(cf);          //执行模块解析配置前的回调函数
}

cf->module_type = NGX_STREAM_MODULE;       //设置模块的类型标志量
cf->cmd_type = NGX_STREAM_MAIN_CONF;       //设置命令的类型标志量

rv = ngx_conf_parse(cf, NULL);            //使用新的 ctx 递归解析块内指令

for (m = 0; cf->cycle->modules[m]; m++) {  //遍历模块数组，只处理 stream 模块
    ...                                  //执行模块的 init_main_conf 和 merge_srv_conf，合并配置
}

ngx_stream_init_phases(cf, cmcf);         //初始化处理引擎
```

```
for (m = 0; cf->cycle->modules[m]; m++) { //遍历模块数组, 只处理 stream 模块
    module->postconfiguration(cf); //执行模块解析配置后的回调函数
}
```

```
ngx_stream_variables_init_vars(cf); //初始化 stream 里使用的变量数组
```

```
ngx_stream_optimize_servers(cf, &ports); //添加到 cycle 的监听端口数组
```

函数 `ngx_stream_block()` 的代码逻辑并不难理解, 主要工作就是处理 `cycle->modules` 数组里的 stream 模块, 为它们分配内存空间, 然后调用 `create_main_conf`、`create_srv_conf`、`preconfiguration`、`postconfiguration` 等函数, 初始化每个 stream 模块的配置。

16.1.3 核心模块

模块 `ngx_stream_core_module` 是 stream 框架里的核心功能模块, 它定义了 `server`、`listen`、`resolver` 等重要指令, 在 `ngx_stream_module` 的基础之上搭建起整个 stream 框架。

`ngx_stream_core_module` 使用两个配置结构体: `ngx_stream_core_srv_conf_t` 和 `ngx_stream_core_main_conf_t`, 存储了整个 `stream{}` 的所有信息。

`ngx_stream_core_srv_conf_t`

`ngx_stream_core_srv_conf_t` 保存的是 `server{}` 块的信息, 定义摘要如下:

```
// 位于 stream/nginx_stream.h
typedef struct {
    ngx_stream_content_handler_pt handler; //内容处理函数
    ngx_stream_conf_ctx_t* ctx; //配置结构体数组

    ngx_flag_t tcp_nodelay; //tcp_nodelay 特性
    size_t preread_buffer_size; //预读缓冲区大小
    ngx_msec_t preread_timeout; //预读的超时时间

    ngx_msec_t resolver_timeout; //域名解析的超时时间
    ngx_resolver_t* resolver; //域名解析对象指针

    ngx_uint_t listen; //是否定义了监听端口
} ngx_stream_core_srv_conf_t;
```

结构体里有两个关键成员：handler 和 ctx。

handler 是此 server 的内容处理函数，作用与 http 模块 loc_conf 里的 handler 类似，用来响应客户端请求，提供服务。但不同的是每个 server 必须定义一个有效的 handler，否则 Nginx 会报错（因为不可能为 TCP/UDP 协议提供通用的默认处理函数）。

ctx 的类型是 ngx_stream_conf_ctx_t，它保存的是所有 stream 模块在 server{} 层次的配置。

ngx_stream_core_main_conf_t

ngx_stream_core_main_conf_t 管理整个 Stream 机制，里面的成员都非常重要，定义摘要如下：

```
// 位于 stream/nginx_stream.h
typedef struct {
    ngx_array_t          servers;           //存储 ngx_stream_core_srv_conf_t*
    ngx_array_t          listen;           //存储 ngx_stream_listen_t

    ngx_stream_phase_engine_t phase_engine; //处理引擎

    ngx_array_t          variables;         //存储所有的变量对象

    ngx_stream_phase_t   phases[NGX_STREAM_LOG_PHASE + 1]; //阶段数组
} ngx_stream_core_main_conf_t;
```

成员 servers 是一个动态数组，保存在 stream{} 里定义的所有 server{} 信息，也就是 ngx_stream_core_srv_conf_t 指针，在解析 server{} 时 Nginx 会添加进这个数组。

成员 listen 也是一个动态数组，保存所有的监听端口信息。

成员 phase_engine 和 phases 是 Stream 机制的处理引擎，将在 16.3 节介绍。

16.1.4 结构关系图

Stream 模块体系里的结构关系图如图 16-1 所示。

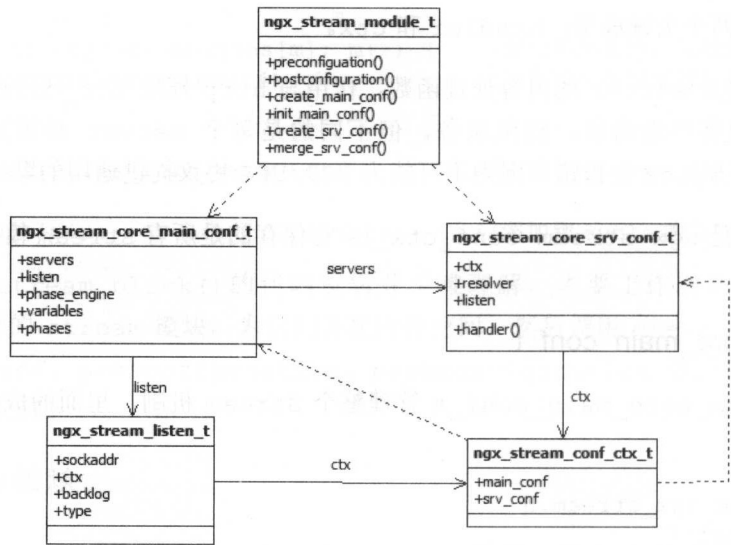


图 16-1 Stream 模块体系的结构关系

16.1.5 存储模型

在配置解析完成后，Stream 模块配置数据的存储模型如图 16-2 所示。

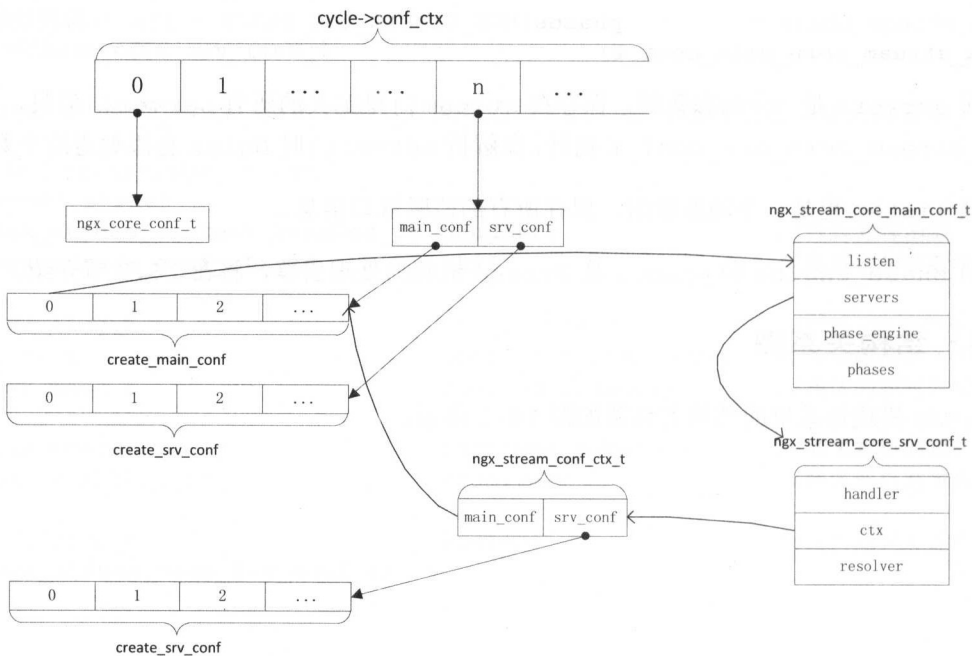


图 16-2 Stream 模块配置数据的存储模型

16.2 监听端口

Nginx 作为一个 Web Server，最重要的工作之一就是定义监听的端口，只有正确配置了监听端口的参数才能对外提供服务。

Nginx 对监听端口的处理比较复杂，定义了许多相关的数据结构，并且与进程机制、事件机制都有互动，需要仔细研究源码才能梳理清楚逻辑流程。

16.2.1 结构定义

Stream 框架里与监听端口相关的结构主要有：^①

- ngx_stream_listen_t;
- ngx_stream_conf_port_t;
- ngx_stream_conf_addr_t;
- ngx_stream_port_t;
- ngx_stream_in_addr_t;
- ngx_stream_addr_conf_t。

这些结构体的名字和功能比较相似，本节将逐个讲解。

ngx_stream_listen_t

14.4.3 节里介绍的 ngx_listening_t 代表的是最基本的 socket 系统调用 listen() 和 accept()，而在 stream 框架里又定义了结构体 ngx_stream_listen_t，用来记录更具体的监听端口信息。

ngx_stream_listen_t 的定义摘要如下：

```
// 位于 stream/ngx_stream.h
typedef struct {
    ngx_sockaddr_t      sockaddr;    //socket 地址，使用 union 适应各种情形
    socklen_t           socklen;     //socket 地址长度

    ngx_stream_conf_ctx_t* ctx;      //监听端口所在的 server{} 配置数组

    unsigned             bind:1;     //要求绑定
    unsigned             wildcard:1; //使用通配符的标志位
    unsigned             ssl:1;      //使用 SSL 的标志位
}
```

^① 这里暂不考虑 ipv6 相关的结构定义。

```

unsigned                ipv6only:1;                //IPV6 的标志位
unsigned                reuseport:1;                //启用 reuseport 特性
unsigned                so_keepalive:2;            //启用 so_keepalive 特性
int                    backlog;                    //内核里等待连接的队列长度
int                    type;                        //socket 的类型, SOCK_STREAM 表示 TCP
} ngx_stream_listen_t;

```

`ngx_stream_listen_t` 的结构比较简单, 它直接对应到 Nginx 配置文件里的 `listen` 指令, 保存了监听端口的配置参数, 并最终把这些参数赋值给 `ngx_listening_t`。

成员 `sockaddr` 表示监听的 IP 地址, 它实际上是一个 `union`, 内部支持 `ipv4/ipv6/unix` 等形式的地址, 通常我们使用的是 `ipv4` 的 `sockaddr_in`。

`ctx` 是结构体里非常关键的成员, 它保存的是定义 `listen` 指令的 `server{}` 的配置数组, 也就是说通过它就可以获取所有 `stream` 模块在这个 `server{}` 里的配置信息。

`ngx_stream_conf_port_t/ngx_stream_conf_addr_t`

`ngx_stream_conf_port_t` 用于整理在 `stream{}` 里定义的监听端口, 定义是:

```

// 位于 stream/nginx_stream.h
typedef struct {
    int                family;                //端口使用的协议族
    int                type;                  //端口的协议类型, TCP/UDP
    in_port_t          port;                  //端口号
    ngx_array_t        addrs;                //端口对应的多个地址
} ngx_stream_conf_port_t;

```

因为在配置里可能会有多个 `server` 监听相同端口的情况出现, 所以 Nginx 使用 `ngx_stream_conf_port_t` 来合并这些相同的端口, 不同的地址存储在 `addrs` 成员。

`addrs` 是一个动态数组, 里面的元素是 `ngx_stream_conf_addr_t`, 但实际上就是 `ngx_stream_listen_t`, 结构很简单, 只是个单纯的包装:

```

typedef struct {
    ngx_stream_listen_t    opt;                //使用 opt 字段来保存端口信息, 里面有地址
} ngx_stream_conf_addr_t;

```

`ngx_stream_port_t/ngx_stream_in_addr_t/ngx_stream_addr_conf_t`

`ngx_stream_port_t` 是一个简单的数组, 存储在 `ngx_listening_t.servers` 里, 实现事件机制里监听端口到 Stream 机制的关联:

```

// 位于 stream/nginx_stream.h
typedef struct {
    void*                addrs;                //存储 ngx_stream_in_addr_t, 即地址
}

```

```
    ngx_uint_t          naddrs;          //数组的长度
} ngx_stream_port_t;

typedef struct {
    in_addr_t           addr;             //IP 地址, 通常是 32 位的整数
    ngx_stream_addr_conf_t conf;          //该地址对应的配置信息
} ngx_stream_in_addr_t;

typedef struct {
    ngx_stream_conf_ctx_t* ctx;           //监听端口所在的配置结构体数组
    ngx_str_t            addr_text;       //监听地址的文本形式
    unsigned              ssl:1;          //使用 SSL 的标志位
} ngx_stream_addr_conf_t;

    ngx_stream_port_t.addrs 是一个数组, 里面存放一个或多个 ngx_stream_in_
    addr_t 对象, 记录了“端口=>地址=>server{}配置”的信息。
```

结构关系图

Nginx 监听端口使用的结构体关系如图 16-3 所示。

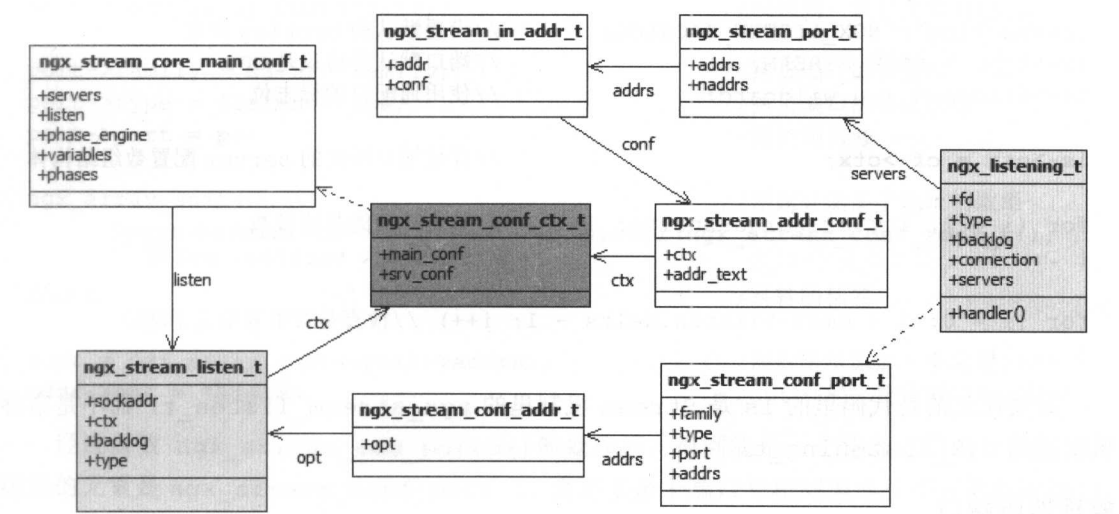


图 16-3 Nginx 监听端口使用的结构体关系

16.2.2 解析配置

在配置解析阶段, Nginx 对监听端口的处理可以分为以下三个步骤:

- 1) 解析“listen”指令, 创建 ngx_stream_listen_t 对象;
- 2) 整理所有监听端口, 创建 ngx_stream_conf_port_t 数组;

3) 对已经整理好的监听端口数组排序, 创建 `ngx_listening_t` 对象。

解析指令

当 Nginx 在配置文件里遇到指令 “listen” 时, 就会调用函数 `ngx_stream_core_listen()` 解析监听端口参数, 保存在 `ngx_stream_listen_t` 对象里, 并添加到监听端口数组 `cmcf->listen`, 代码摘要如下:

```
// 位于 stream/nginx_stream_core_module.c
cscf->listen = 1; //表示 server 已经定义了监听端口

value = cf->args->elts; //取配置参数数组
u.url = value[1]; //第一个字符串是端口地址
ngx_parse_url(cf->pool, &u); //解析地址

ls = ngx_array_push(&cmcf->listen); //添加到监听端口数组

ngx_memcpy(&ls->sockaddr.sockaddr, &u.sockaddr, u.socklen);
ls->socklen = u.socklen; //拷贝地址信息

ls->backlog = NGX_LISTEN_BACKLOG; //设置默认的 backlog 参数
ls->type = SOCK_STREAM; //端口默认的协议是 TCP
ls->wildcard = u.wildcard; //使用通配符的标志位

ls->ctx = cf->ctx; //存储端口所在的 server 配置数组结构体

for (i = 2; i < cf->args->nelts; i++) //检查端口的其他参数
{ ... } //例如 udp、backlog、ssl 等

for (i = 0; i < cmcf->listen.nelts - 1; i++) //检查是否重复定义了端口
{ ... }
```

需要注意的是代码里的 `ls` 是 Stream 机制里的 `ngx_stream_listen_t`, 而不是事件机制里的 `ngx_listening_t`。

整理监听端口

`stream{}` 配置解析完成后, 函数 `ngx_stream_block()` 创建一个临时使用的 `ngx_stream_conf_port_t` 数组, 对 `cmcf->listen` 里的端口进行归并整理:

```
// 位于 stream/nginx_stream.c
ngx_array_init(&ports, //一个临时使用的数组
    cf->temp_pool, 4, sizeof(ngx_stream_conf_port_t);

listen = cmcf->listen.elts; //监听端口所在的数组
```

```
for (i = 0; i < cmcf->listen.nelts; i++) { //遍历数组
    ngx_stream_add_ports(cf, &ports, &listen[i]); //对相同的监听端口进行合并处理
}
```

函数 ngx_stream_add_ports() 把相同的监听端口都存储在一个 `addrs` 数组里:

```
sa = &listen->sockaddr.sockaddr; //取监听的地址
p = ngx_inet_get_port(sa); //取监听的端口号

port = ports->elts; //数组首地址
for (i = 0; i < ports->nelts; i++) { //遍历数组, 查找是否有相同端口
    if (p == port[i].port //要求端口号相同
        && listen->type == port[i].type //协议类型相同
        && sa->sa_family == port[i].family) //协议族相同
    {
        port = &port[i]; //找到了一个相同的端口定义
        goto found; //跳转, 执行合并动作
    }
}

port = ngx_array_push(ports); //没找到, 就是个新端口

port->family = sa->sa_family; //拷贝端口的协议族
port->type = listen->type; //拷贝端口的协议类型
port->port = p; //拷贝端口号

ngx_array_init( //创建存储多个地址的数组
    &port->addrs, cf->temp_pool, 2, sizeof(ngx_stream_conf_addr_t));

found: //跳转的标签

addr = ngx_array_push(&port->addrs); //向数组里添加一个元素
addr->opt = *listen; //opt 存储完整的端口配置信息
```

经过函数 ngx_stream_add_ports() 的处理后, 得到的是一个整理好的端口数组, 数组里的元素是 ngx_stream_conf_port_t, 合并了多个端口号相同但地址不同的监听端口。

创建监听端口

函数 ngx_stream_optimize_servers() 使用整理好的端口数组, 调用 ngx_create_listening() 创建用于 Nginx 事件机制的监听对象 ngx_listening_t, 代码摘要如下:

```
// 位于 stream/nginx_stream.c
port = ports->elts; //数组首地址
for (p = 0; p < ports->nelts; p++) { //遍历数组, 创建监听对象
    ngx_sort(...); //对地址排序, 决定优先级
```

```

addr = port[p].addrs.elts;                                //取此端口的地址数组

while (i < last) {                                        //遍历端口的地址数组, 逐个创建
    ls = ngx_create_listening(                            //创建监听对象, 添加进 cycle
        cf, &addr[i].opt.sockaddr.sockaddr, addr[i].opt.socklen);

    ls->handler = ngx_stream_init_connection;              //设置接受连接的回调函数
    ls->pool_size = 256;                                   //内存池大小是 256 字节
    ls->type = addr[i].opt.type;                           //设置监听的协议类型
    ls->backlog = addr[i].opt.backlog;                     //设置监听的 backlog

    stport = ngx_palloc(                                   //创建关联的端口信息数组
        cf->pool, sizeof(ngx_stream_port_t));

    ls->servers = stport;                                  //关联到 servers 字段
    ngx_stream_add_addrs(cf, stport, addr);               //把地址信息拷贝进数组
}                                                         //while 处理完一个端口
}                                                         //for 处理完所有端口

```

函数 `ngx_stream_optimize_servers()` 里的关键操作是设置接受连接时的回调函数 `ngx_stream_init_connection`, 根据 14.8.10 节的分析, 当 Nginx 事件机制处理 `accept` 事件时会调用 `ngx_event_accept()` 或 `ngx_event_recvmsg()`, 创建连接对象, 并调用 `ls->handler`, 从而进入 Stream 机制的工作流程。

`ls->servers` 里存储了监听端口对应的多个地址, 以及它们所在的 `server{}` 信息, 通过它 Nginx 可以定位到具体的 `server`, 调用 `content handler` 处理。

解析流程图

Nginx 解析配置指令, 添加监听端口的流程图如图 16-4 所示。

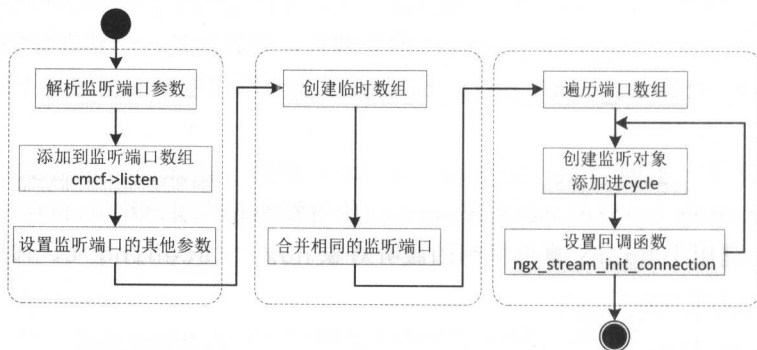


图 16-4 添加监听端口的流程

16.2.3 启动监听

14.8.5 节从事件机制的角度描述了 Nginx 监听端口的初始化, 这里我们再结合 Stream 机制更详细地介绍监听的启动过程。

stream{} 配置解析完成之后, cycle->listening 里就保存了所有监听端口的信息, 此时 Nginx 仍然处在初始化阶段的 ngx_init_cycle() 函数里, 它执行 ngx_open_listening_sockets() 和 ngx_configure_listening_sockets() 两个函数正式启动监听。

打开监听端口

函数 ngx_open_listening_sockets() 调用系统函数 socket/bind/listen 启动监听, 核心代码摘要如下:

```
// 位于 core/nginx_connection.c
ls = cycle->listening.elts; // 监听对象数组首地址
for (i = 0; i < cycle->listening.nelts; i++) { // 遍历监听对象数组
    if (ls[i].ignore) { // 如果要求忽略则不监听
        continue;
    }

    s = ngx_socket( // 创建 socket
        ls[i].sockaddr->sa_family, ls[i].type, 0);

    setsockopt(s, SOL_SOCKET, SO_REUSEADDR, // 设置 socket 选项
        (const void *) &reuseaddr, sizeof(int));

    bind(s, ls[i].sockaddr, ls[i].socklen); // 绑定 socket 地址

    if (ls[i].type != SOCK_STREAM) { // 检查是否是 TCP 协议
        ls[i].fd = s; // 如果不是 TCP, 即 UDP, 那么无须监听
        continue; // 继续处理下一个监听端口
    }

    listen(s, ls[i].backlog); // 是 TCP, 需要调用 listen 监听
    ls[i].listen = 1; // 设置监听标志位, 已经开始监听
    ls[i].fd = s; // 设置监听的文件描述符, 即 socket
} // for 遍历完监听对象数组后结束
```

这段代码里省略了很多错误处理和其他的一些参数设置, 只保留了最基本的部分, 可以看到, Nginx 的启动监听并没有什么“魔法”, 与通常的服务器程序没有太多不同。

设置并启动监听端口

函数 ngx_open_listening_sockets() 仅对监听端口做了最基本的设置 (backlog、SO_REUSEADDR、SO_REUSEPORT 等), Nginx 把大部分的监听端口设置放在了 ngx_configure_listening_sockets() 里, 设置 SO_RCVBUF、SO_SNDBUF、SO_KEEPALIVE、TCP_FASTOPEN、TCP_DEFER_ACCEPT 等选项, 代码都很容易理解, 故不再列出。

监听端口设置完成之后, Nginx 继续执行 ngx_init_cycle() 函数的剩余部分, 然后 fork 出 worker 进程运行函数 ngx_worker_process_cycle() (可参考 13.6 节)。

事件机制在 worker 进程初始化时创建了连接池, 从连接池里获取空闲连接对象, 关联到监听端口, 把端口 socket 的读事件加入 epoll 监控, 最后进入无限 for 循环收集处理事件。

直到这时, Nginx 才真正开始对外提供服务。

流程图

Nginx 启动监听的流程如图 16-5 所示。

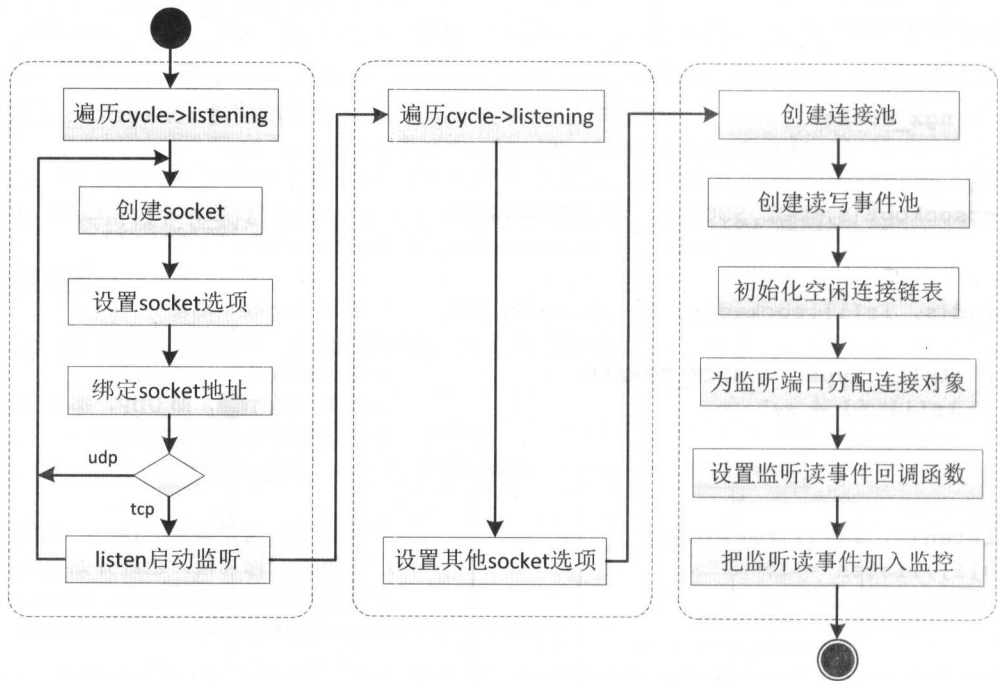


图 16-5 Nginx 启动监听的流程

16.3 处理引擎

Stream 机制也有处理阶段的概念，提供与 HTTP 机制类似的处理引擎，能够灵活方便地介入 TCP/UDP 协议处理的各个阶段。^①

16.3.1 阶段定义

目前 Nginx 只为 Stream 机制划分出了 7 个处理阶段，比 HTTP 机制的 11 个要少：

```
typedef enum {
    NGX_STREAM_POST_ACCEPT_PHASE = 0,           //接受连接之后
    NGX_STREAM_PREACCESS_PHASE,                 //检查访问权限前
    NGX_STREAM_ACCESS_PHASE,                   //检查访问权限
    NGX_STREAM_SSL_PHASE,                      //SSL 处理
    NGX_STREAM_PREREAD_PHASE,                  //预读数据
    NGX_STREAM_CONTENT_PHASE,                 //产生响应内容
    NGX_STREAM_LOG_PHASE                       //会话结束后记录日志
} ngx_stream_phases;
```

Stream 机制里的这些阶段不存在限制，模块可以加载到任意的阶段里运行，但 content 阶段是个例外，一个 server{} 只能有一个 content handler。

preread 阶段是 Stream 机制的独有处理阶段，顾名思义，它从客户端“预读”部分数据，可以用来在正式处理之前解析信息，例如 SNI 协议头。

16.3.2 函数原型

Nginx 为 stream 模块定义了两个处理函数原型：

```
// 位于 stream/nginx_stream.h
typedef ngx_int_t (*ngx_stream_handler_pt) (ngx_stream_session_t *s);
typedef void (*ngx_stream_content_handler_pt) (ngx_stream_session_t *s);
```

ngx_stream_handler_pt 是通用的阶段处理函数，可以在 access/preread 等阶段里处理数据，而 ngx_stream_content_handler_pt 是仅用在 content 阶段的处理函数，也就是 ngx_stream_core_srv_conf_t 里的 handler。

两个函数的定义很像，区别在于返回值，content_handler 的返回类型是 void。

^① 早期的 stream 模块没有很完善的处理引擎，只是提供了几个很简单的 access、limit_conn handler，直至 1.11.5 才变成了现在的处理引擎。

16.3.3 处理函数的存储方式

Stream 机制使用 `ngx_stream_phase_t` 存储每个阶段可用的处理函数：

```
// 位于 stream/nginx_stream.h
typedef struct {
    ngx_array_t      handlers;           //处理函数数组
} ngx_stream_phase_t;
```

结构体 `ngx_stream_core_main_conf_t` 里的成员 `phases` 存储所有的阶段：

```
// 位于 stream/nginx_stream.h
typedef struct {
    ...                                //其他成员
    ngx_stream_phase_t phases[NGX_STREAM_LOG_PHASE + 1]; //阶段数组
} ngx_stream_core_main_conf_t;
```

与 HTTP 机制一样，只要在模块的 `postconfiguration` 函数里向相应的 `phases` 数组添加元素就可以把函数注册到 Stream 处理流程里。

16.3.4 引擎数据结构

Stream 机制里的处理引擎也与 HTTP 机制的类似，使用阶段 checker 来间接调用阶段 handler，把 `phases` 二维数组用 `next` 指针组织为一个一维数组，实现阶段的灵活跳转：

```
//阶段处理数据结构
typedef struct ngx_stream_phase_handler_s ngx_stream_phase_handler_t;

//checker 的函数原型
typedef ngx_int_t (*ngx_stream_phase_handler_pt)(
    ngx_stream_session_t *s, ngx_stream_phase_handler_t *ph);

struct ngx_stream_phase_handler_s {           //阶段处理结构体
    ngx_stream_phase_handler_pt checker;       //检查器函数
    ngx_stream_handler_pt handler;           //模块的处理函数
    ngx_uint_t next;                         //下一处理阶段的序号
};

typedef struct {
    ngx_stream_phase_handler_t* handlers;     //包含所有模块的 handler
} ngx_stream_phase_engine_t;                 //处理引擎定义
```

结构体 `ngx_stream_core_main_conf_t` 里的成员 `phase_engine` 就是 Stream 机制的处理引擎：

```
typedef struct {
    ...
    ngx_stream_phase_engine_t    phase_engine;           //其他成员
                                                    //阶段处理引擎数组
} ngx_stream_core_main_conf_t;
```

16.3.5 结构关系图

Stream 机制处理引擎用到的数据结构关系如图 16-6 所示。

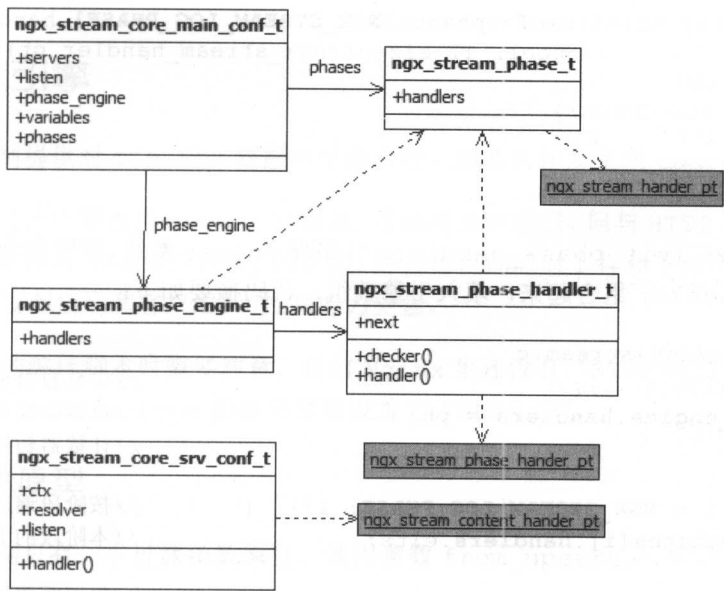


图 16-6 Stream 机制处理引擎的数据结构关系

16.3.6 引擎的初始化

Stream 处理引擎的初始化发生在 Nginx 配置解析阶段。

函数 ngx_stream_block() 在解析完 stream{} 后，依次调用 ngx_stream_init_phases() 和 ngx_stream_init_phase_handlers() 这两个函数生成处理引擎。

ngx_stream_init_phases() 使用 ngx_array_init() 初始化 phases 数组，但因为只能有一个 content handler，所以不包含 content 阶段，代码摘要如下：

```
// 位于 stream/nginx_stream.c
static ngx_int_t ngx_stream_init_phases(...)
{
    if (ngx_array_init(&cmcf->phases[NGX_STREAM_POST_ACCEPT_PHASE].handlers,
```

```

        cf->pool, 1, sizeof(ngx_stream_handler_pt))
    != NGX_OK)
{
    return NGX_ERROR;
}

... //其他阶段的初始化

if (ngx_array_init(&cmcf->phases[NGX_STREAM_PREREAD_PHASE].handlers,
    cf->pool, 1, sizeof(ngx_stream_handler_pt))
    != NGX_OK)
{
    return NGX_ERROR;
}

if (ngx_array_init(&cmcf->phases[NGX_STREAM_LOG_PHASE].handlers,
    cf->pool, 1, sizeof(ngx_stream_handler_pt))
    != NGX_OK)
{
    return NGX_ERROR;
}

return NGX_OK;
}

```

ngx_stream_init_phase_handlers() 整理 phases 数组, 按阶段分类, 把 handler 与对应阶段的 checker 组合起来, 填入引擎数组, 代码摘要如下:

```

// 位于 stream/nginx_stream.c
ph = ngx_palloc(...); //引擎数组分配内存
cmcf->phase_engine.handlers = ph; //设置处理引擎
n = 0; //计数器初始化

for (i = 0; i < NGX_STREAM_LOG_PHASE; i++) { //按阶段遍历, 注意不含 log
    h = cmcf->phases[i].handlers.elts; //本阶段的 handler 数组

    switch (i) { //根据阶段选择 checker

    case NGX_STREAM_PREREAD_PHASE: //preread 阶段
        checker = ngx_stream_core_preread_phase; //该阶段使用的 checker
        break;

    case NGX_STREAM_CONTENT_PHASE: //content 阶段
        ph->checker = ngx_stream_core_content_phase; //该阶段使用的 checker
        n++; //content handler 只有一个, 所以不需要再检查
        ph++; //直接跳到下一个阶段, 也就是 log
        continue;

    default:
        checker = ngx_stream_core_generic_phase; //默认的 checker
    }

    n += cmcf->phases[i].handlers.nelts; //该阶段的 handler 总数

    //反向遍历 phases 数组, 向引擎数组添加元素
    for (j = cmcf->phases[i].handlers.nelts - 1; j >= 0; j--) {

```

```

    ph->checker = checker;                //设置 checker
    ph->handler = h[j];                    //设置 handler
    ph->next = n;                          //设置阶段跳转序号
    ph++;
}
//phases 数组遍历结束

```

函数执行之后，`phase_engine.handlers` 数组就把所有 stream 模块的 handler 组成了一维数组，数组元素（`ngx_stream_phase_handler_t` 结构）里的 `next` 形成了静态链表，指示阶段跳转的数组序号，可以跳过本阶段的其他 handler，直接进入下个处理阶段。

16.4 过滤引擎

Stream 机制提供对 TCP/UDP 数据的过滤功能，过滤动作发生在 content 阶段。

Stream 的过滤引擎也是 output 过滤器，过滤发出的数据，但与 HTTP 的过滤引擎不同，Stream 过滤引擎没有 header/body 的区别，并且可以同时对上行和下行两个方向的数据执行过滤操作（即发到后端和客户端），控制能力更强。

在编写模块的编译脚本时需要注意，目前 Nginx 里还没有“`STREAM_FILTER`”的类型，Shell 变量 `ngx_module_type` 的值仍然要设置为“`STREAM`”。

16.4.1 函数原型

Stream 机制只有一个过滤函数原型，使用参数 `from_upstream` 标记数据的来源：

```

// 位于 stream/ngx_stream.h
typedef ngx_int_t (*ngx_stream_filter_pt) (
    ngx_stream_session_t *s, ngx_chain_t *chain, ngx_uint_t from_upstream);

```

在过滤函数里可以根据入口参数 `from_upstream` 判断数据是上行还是下行，决定处理数据的具体行为。

16.4.2 过滤函数链表

Nginx 在 `stream/ngx_stream.c` 里定义了过滤函数链表的头节点：

```

ngx_stream_filter_pt ngx_stream_top_filter;    //过滤函数链表的头节点

```

与 HTTP 过滤引擎一样，在使用过滤函数链表时也要在 `postconfiguration` 阶段用模块内部的静态变量 `next_filter` 保存 `ngx_stream_top_filter`，然后设置 `ngx_stream_top_filter` 为模块自己的过滤函数，把函数添加到链表的头节点。

16.5 运行机制

Stream 机制处理的是 TCP/UDP 原始字节流, 数据内容完全是不透明的, 无法预知格式, 所以 Nginx 框架能做的工作不是太多, Stream 处理引擎只能提供一些基本的功能, 更多的逻辑必须要我们结合事件机制自己手工处理, 虽然有些麻烦但也带来了更多的自由。

16.5.1 会话结构体

Stream 机制使用结构体 `ngx_stream_session_t` 表示 TCP/UDP 通信过程, 它的地位、作用与 HTTP 机制里的 `ngx_http_request_t` 类似, 是处理 TCP/UDP 协议的核心数据结构。

`ngx_stream_session_t` 的定义摘要如下:

```
// 位于 stream/nginx_stream.h
typedef struct ngx_stream_session_s  ngx_stream_session_t;

struct ngx_stream_session_s {
    uint32_t                signature;                // "STRM" 的 ASCII 码

    ngx_connection_t*       connection;               // 客户端的连接对象

    off_t                   received;                  // 收到的字节数
    time_t                  start_sec;                  // 开始时间, 秒
    ngx_msec_t              start_msec;                // 开始时间, 毫秒

    void**                  ctx;                       // 处理请求的环境数据
    void**                  main_conf;                  // main 层次的配置信息数组
    void**                  srv_conf;                   // srv 层次的配置信息数组

    ngx_stream_variable_value_t* variables;           // 存储变量值的数组

    ngx_int_t               phase_handler;             // 当前处理阶段的索引号
    ngx_uint_t              status;                    // 处理结果的状态码

    unsigned                ssl:1;                     // 暂未使用
    unsigned                stat_processing:1;         // 暂未使用
    unsigned                health_check:1;            // 暂未使用
};
```

成员 `connection` 是会话结构体里的关键, 它表示了 Nginx 与客户端的连接, 我们必须处理它上面的读写事件才能完成数据的收发工作。

结构体里的其他成员与 `ngx_http_request_t` 类似: `ctx` 是会话的环境数据, 用来暂存中间结果; `main_conf` 和 `srv_conf` 分别保存了 `stream` 模块在配置文件中对应的配置信

息；phase_handler 标记了在当前处理过程中所在阶段的 handler 序号，用于执行引擎。

要注意的一点是会话结构体里没有内存池成员，如果要分配会话期间使用的内存必须使用连接对象的内存池，即 connection->pool。

ngx_stream_session_t 与 Nginx 框架里其他结构体的关系如图 16-7 所示。

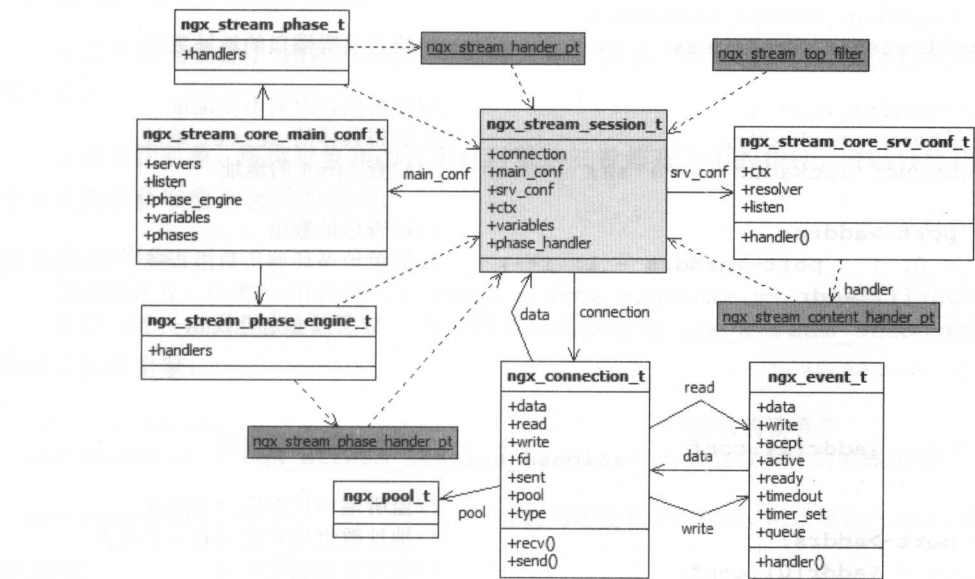


图 16-7 ngx_stream_session_t 与 Nginx 框架里其他结构体的关系

16.5.2 创建会话

我们先回顾一下事件机制（14.8.10 节）：

在事件机制的 ngx_process_events_and_timers() 函数里 Nginx 处理事件，对监听端口的可读事件调用 ngx_event_accept() 接受一个连接，从连接池里获得空闲连接对象并为它设置了基本的参数，然后调用 ls->handler，也就是 ngx_stream_init_connection()，正式进入 Stream 处理机制。

注意，虽然此时我们得到了客户端连接对象，但它的读事件和写事件并没有加入 epoll 监控，需要后续的 stream 模块自己根据实际情况注册读写事件的关联。

函数 ngx_stream_init_connection() 的工作要点有三个：

- 查找对应的 server{} 配置信息；
- 创建会话对象 ngx_stream_session_t；

■ 启动处理引擎。

查找 server 信息

`c->listening->servers` 里存储了监听端口的地址信息，如果是多个地址需要根据连接的本地地址在 `addrs` 数组查找，否则可直接获取对应的 `server{}` 配置信息：

```
// 位于 stream/nginx_stream_handler.c
port = c->listening->servers; // 检查监听端口的地址数组

if (port->naddrs > 1) { // 监听端口对应多个地址
    sa = c->local_sockaddr; // 获得具体的 server 地址
    sin = (struct sockaddr_in *) sa; // 检查 ipv4 的地址

    addr = port->addrs; // 检查地址数组
    for (i = 0; i < port->naddrs - 1; i++) { // 简单地循环遍历数组查找
        if (addr[i].addr == sin->sin_addr.s_addr) { // 因为有 * 通配符 (在数组末尾)
            break; // 所以必定会找到一个
        }
    }

    addr_conf = &addr[i].conf; // 取对应的配置信息
} else { // 监听端口只对应一个地址
    addr = port->addrs; // 地址数组里肯定只有一个元素
    addr_conf = &addr[0].conf; // 取对应的配置信息
}
```

目前 Nginx 在查找同端口号的多个不同地址时使用的是简单的循环遍历，效率不是很高，也许将来会改成更高效的散列表方式。

创建会话对象

确定了配置信息也就确定了用来处理连接的 `server{}` 块还有里面的 `handler`，使用这些信息就可以创建会话对象：

```
s = ngx_palloc( // 创建会话对象
    c->pool, sizeof(ngx_stream_session_t));

s->signature = NGX_STREAM_MODULE; // 设置对象的签名
s->main_conf = addr_conf->ctx->main_conf; // 关联 main 配置数组
s->srv_conf = addr_conf->ctx->srv_conf; // 关联 srv 配置数组

s->connection = c; // 会话对象关联连接对象
c->data = s; // 连接对象再关联会话对象
```



```
s->ctx = ngx_palloc( //创建会话使用的 ctx 数据
    c->pool, sizeof(void *) * ngx_stream_max_module);

s->variables = ngx_palloc(...); //创建会话的变量值数组

tp = ngx_timeofday(); //获取当前时间
s->start_sec = tp->sec; //设置会话的开始时间
s->start_msec = tp->msec;
```

会话对象里包含了连接、配置、变量等大量的信息，stream 模块使用它就可以处理客户端的请求。

注意会话对象、连接对象和读写事件对象这三者都有互指的指针，所以只要持有任意一个对象就能够获取整个会话的信息。

启动处理引擎

最后，Stream 机制设置连接上读事件的处理函数为 ngx_stream_session_handler，启动了处理引擎：

```
rev = c->read; //会话的读事件
rev->handler = ngx_stream_session_handler; //设置读事件的处理函数

rev->handler(rev); //因为还没有事件触发，所以要自己调用，即“主动触发”
```

流程图

Stream 机制创建会话的流程图如图 16-8 所示。

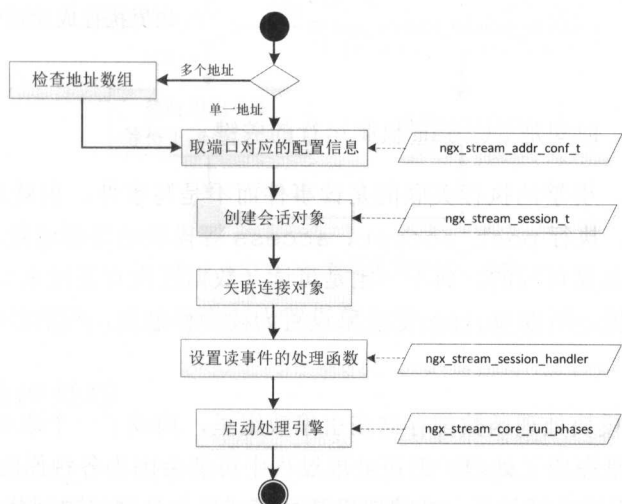


图 16-8 Stream 机制创建会话的流程

16.5.3 执行引擎

在函数 `ngx_stream_init_connection()` 里设置了读事件的处理函数 `ngx_stream_session_handler`，它就是 Stream 机制处理引擎的启动器，只要 Nginx 监控到连接上有读事件（收到数据或断连）时就会调用它执行引擎。

函数 `ngx_stream_session_handler()` 的代码很简单，主要是从读事件的指针里重新获取会话对象，然后执行引擎函数 `ngx_stream_core_run_phases()`：

```
// 位于 stream/ngx_stream_handler.c
void ngx_stream_session_handler(ngx_event_t *rev) //读事件处理函数
{
    c = rev->data;                                //从读事件里获得连接对象
    s = c->data;                                    //从连接对象里获得会话对象

    ngx_stream_core_run_phases(s);                //使用会话对象运行引擎
}
```

`ngx_stream_core_run_phases()` 使用会话对象里的成员 `phase_handler` 作为处理引擎的索引，执行引擎数组里的 `checker` 函数，也就是执行了 `stream` 模块的 `handler`：

```
ph = cmcf->phase_engine.handlers;                //获取引擎数组首地址

while (ph[s->phase_handler].checker) {           //遍历引擎数组

    rc = ph[s->phase_handler].checker(            //执行 checker，间接调用 handler
        s, &ph[s->phase_handler]);
    if (rc == NGX_OK) {                           //检查返回值
        return;                                    //如果执行成功则暂时退出引擎处理
    }
}
```

代码虽然很简单，但却是 Stream 机制运行的关键。

首先我们要看到，引擎的执行关联的是读事件而不是写事件，也就是说只有当 `socket` 可读时才会启动引擎，执行 `post_accept`、`access` 等模块的处理函数。这是因为当客户端建立连接时 `socket` 总是可写的，而不一定是可读（数据还没有发过来），而且作为服务器首先要从客户端接收数据，所以 Nginx 把引擎设置为读事件触发，写事件需要各个模块自己设置监控和处理函数。

所有的 `stream` 模块处理函数都存储在引擎数组里，构成了一个职责链，TCP/UDP 会话走完这个职责链就算是完成了处理。但在处理过程中可能会因为各种原因（`socket` 不可读/写、等待外部数据）无法继续处理，必须要以某种方式保存处理的中间状态，在下次事件机制

触发时可以在中断的位置继续运行。会话对象里的 `phase_handler` 就扮演了这个“cursor”的角色，保存会话在引擎数组里的位置，指向当前处理阶段暂停的 `stream` 模块。

引擎数组里的 `checker` 函数“包装”了 `stream` 模块的处理函数，它检查模块 `handler` 的返回值。如果返回非 `NGX_OK` 的错误码（通常都是 `NGX_AGAIN`）就表示模块正确处理了会话对象，引擎就可以继续运行。如果是 `NGX_OK`，则表明模块暂无法继续处理，此时会结束循环，控制流程就退出了 `Stream` 机制，又回到了事件机制，当 `socket` 上再发生可读事件时就会再次运行引擎。

`Stream` 处理引擎还有一个隐含的出口：如果模块在处理过程中发生了致命的错误，那么 `Nginx` 会调用函数 `ngx_stream_finalize_session()` 结束会话，直接跳出引擎，不会执行后续的 `stream` 模块。

`Stream` 处理引擎的运行流程如图 16-9 所示。

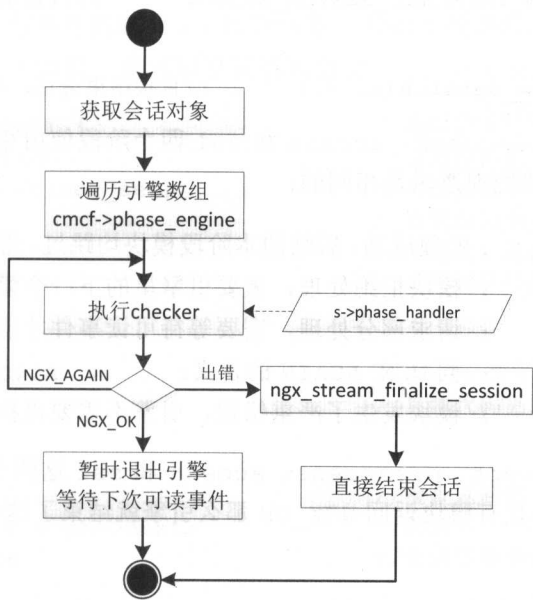


图 16-9 Stream 处理引擎的运行流程

下面我们将逐个讨论 `Stream` 机制里的 `checker` 函数，更进一步地理解引擎的运行过程。

16.5.4 通用阶段处理

函数 `ngx_stream_core_generic_phase()` 管理 `post_accept`、`preaccess`、`access` 和 `ssl` 四个阶段的 `stream` 模块，代码摘要如下：

```

rc = ph->handler(s); //执行模块的处理函数

if (rc == NGX_OK) { //检查返回值，函数执行成功
    s->phase_handler = ph->next; //跳到下一个阶段
    return NGX_AGAIN; //要求引擎继续运行
}

if (rc == NGX_DECLINED) { //检查返回值，此模块拒绝处理
    s->phase_handler++; //使用下一个模块处理会话
    return NGX_AGAIN; //要求引擎继续运行
}

if (rc == NGX_AGAIN || rc == NGX_DONE) { //检查返回值，again或done
    return NGX_OK; //模块暂无法处理请求，等待下次可读事件，要求引擎暂时退出
}

if (rc == NGX_ERROR) { //检查返回值，模块发生了严重错误
    rc = NGX_STREAM_INTERNAL_SERVER_ERROR; //改错误码为 500
}

ngx_stream_finalize_session(s, rc) //直接结束会话，不执行后续的 stream 模块

```

post_accept、preaccess、access 和 ssl 四个阶段使用相同的 checker，这意味着 Stream 机制对它们的处理逻辑是相同的：

- NGX_OK : 处理成功，后续的本阶段模块均跳过，继续进入下一个阶段处理；
- NGX_DECLINED : 模块拒绝处理，需要引擎里的下一个模块处理；
- NGX_AGAIN : 请求部分处理，需要等待可读事件才能继续；
- NGX_DONE : 同 NGX_AGAIN 的含义；
- NGX_ERROR : 模块发生了严重错误，引擎不需要再执行，直接结束会话。

所以，在 post_accept、preaccess、access 和 ssl 这四个阶段里的 stream 模块各自都是“互斥”的，一旦有模块返回 NGX_OK 那么引擎就结束了这个阶段，跳过剩余的模块转到下一个阶段处理。

这四个阶段通常不需要读取数据，使用连接对象里最基本的 IP 地址等信息就足够了。

函数 ngx_stream_core_generic_phase() 的流程图如图 16-10 所示。

16.5.5 预读数据

函数 ngx_stream_core_preread_phase() 负责 Stream 机制的 preread 阶段，读取连接最开始的一小部分数据，可以做预解析工作。

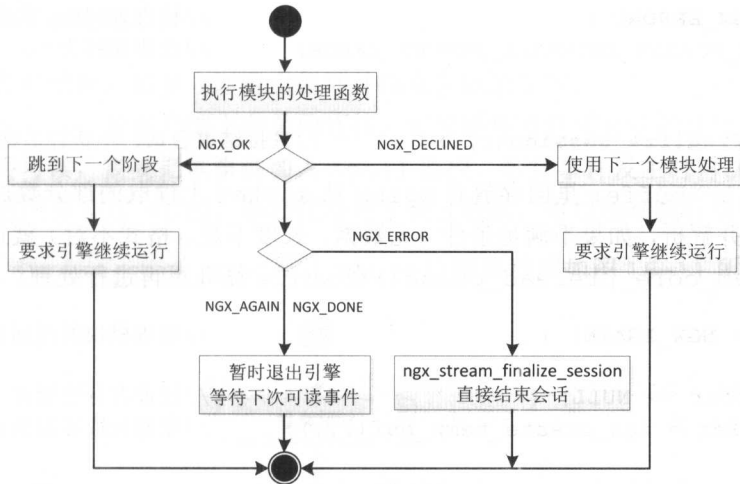


图 16-10 函数 ngx_stream_core_generic_phase() 的流程

由于需要读取数据，与事件机制相关，所以函数一开始必须检查定时器是否已经超时，如果已经超时则表明连接上无数据，就可以直接结束会话：

```
if (c->read->timedout) {                                     //超时仍无可读事件发生
    rc = NGX_STREAM_OK;                                       //设置返回值为 200
} else if (c->read->timer_set) {                               //已经设置了定时器，说明正在等待数据
    rc = NGX_AGAIN;                                           //设置返回值为 again，进入下面的循环
} else {                                                      //未超时，还没有开始预读数据
    rc = ph->handler(s);                                       //调用模块的处理函数
}

while (rc == NGX_AGAIN) {                                     //while 循环预读数据处理
    ...                                                       //具体代码见后
}

if (rc == NGX_OK) {                                           //检查返回值，函数执行成功
    s->phase_handler = ph->next;                               //跳到下一个阶段
    return NGX_AGAIN;                                         //要求引擎继续运行
}

if (rc == NGX_DECLINED) {                                     //检查返回值，此模块拒绝处理
    s->phase_handler++;                                         //使用下一个模块处理会话
    return NGX_AGAIN;                                         //要求引擎继续运行
}

if (rc == NGX_DONE) {                                         //检查返回值，done
    return NGX_OK;                                             //模块暂无法处理请求，等待下次可读事件，要求引擎暂时退出
}
```

```

if (rc == NGX_ERROR) { //检查返回值, 模块发生了严重错误
    rc = NGX_STREAM_INTERNAL_SERVER_ERROR; //改错误码为 500
}

```

```

ngx_stream_finalize_session(s, rc); //直接结束会话, 不执行后续的 stream 模块

```

连接对象的 `c->buffer` 里面存放有 Nginx 从 socket 上读取的部分数据, 在 `preread` 模块里需要检查并解析, 如果不满足条件 (无数据、长度不足、格式不对) 就返回适当的错误码, `ngx_stream_core_preread_phase()` 在 `while` 循环里再进行处理:

```

while (rc == NGX_AGAIN) { //检查模块的返回值, again

    if (c->buffer == NULL) { //还没有分配缓存
        c->buffer = ngx_create_temp_buf(...); //创建读取数据的内存
    }

    if (c->read->eof) { //读到 eof, 客户端关闭连接
        rc = NGX_STREAM_OK; //设置返回值为 200
        break; //结束循环, 之后结束会话
    }

    if (!c->read->ready) { //连接不可读
        ngx_handle_read_event(c->read, 0); //要求监控连接的读事件

        ngx_add_timer(c->read, cscf->preread_timeout); //设置超时时间

        c->read->handler = ngx_stream_session_handler; //设置回调函数为引擎启动器

        return NGX_OK; //模块暂无法处理请求, 等待下次可读事件, 要求引擎暂时退出
    }

    n = c->recv(c, c->buffer->last, size); //连接可读则读取数据

    if (n == NGX_ERROR) { //读取发生错误
        rc = NGX_STREAM_OK; //设置返回值为 200
        break; //结束循环, 之后结束会话
    }

    if (n > 0) { //读取了部分数据
        c->buffer->last += n; //调整缓冲区大小
    }

    rc = ph->handler(s); //调用模块的处理函数再次检查
}

```

在 `while` 循环里 Nginx 会检查连接上的读事件。如果不可读, 那么就把读事件加入 `epoll` 监控并设置超时时间, 返回 `NGX_OK` 要求引擎暂时退出, 等待下次可读事件。如果可读那么就读取数据, 再调用模块的 `handler` 处理。

循环的出口主要有四个：

- 发生各种错误，需要立即退出循环并结束会话的处理；
- 读到了 eof，即客户端主动关闭连接，退出循环并结束会话的处理；
- 连接不可读，需要把读事件加入 `epoll` 监控，等待事件发生时再次运行；
- 模块返回非 `NGX_AGAIN`，即已经处理了预读数据，可以继续执行引擎后续模块。

函数 `ngx_stream_core_preread_phase()` 的流程图如图 16-11 所示。

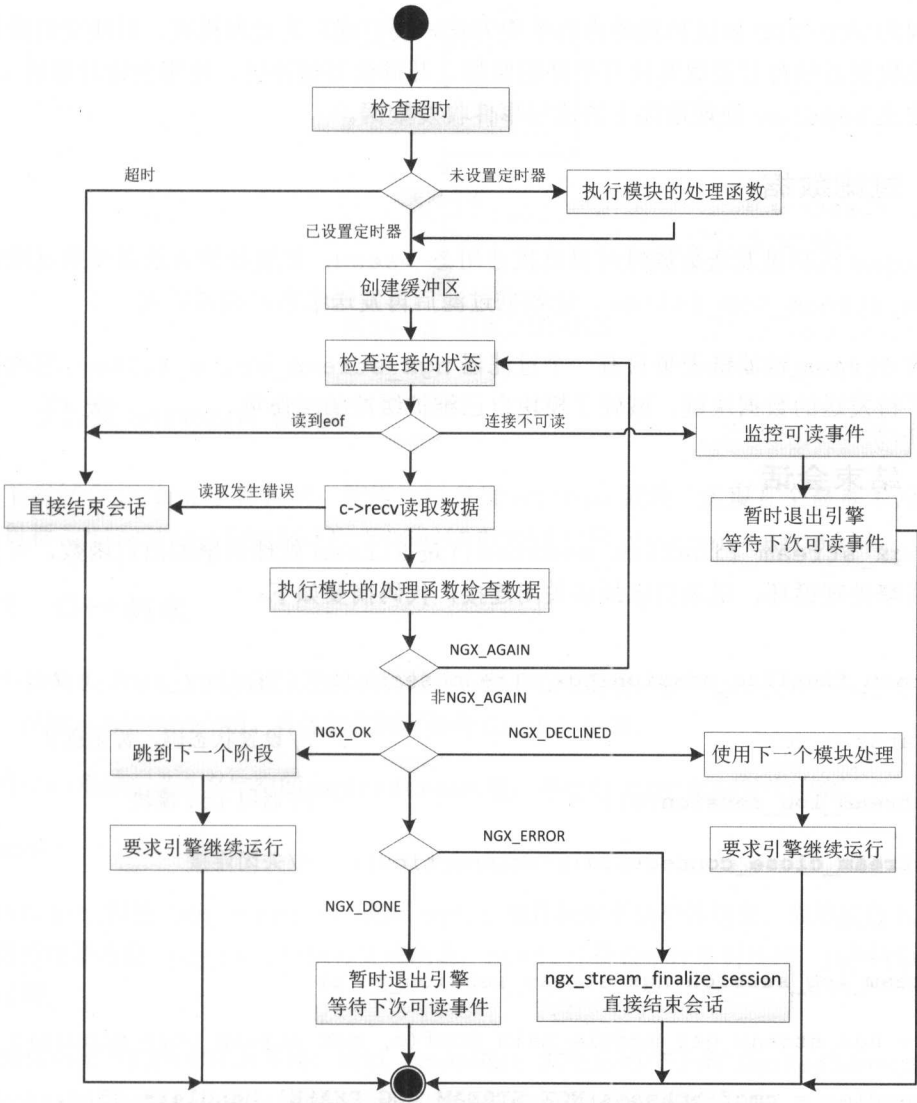


图 16-11 函数 `ngx_stream_core_preread_phase()` 的流程

16.5.6 产生响应数据

content 阶段是产生 TCP/UDP 响应数据的重要阶段,也是处理引擎的最后一个阶段,但 Stream 机制的函数 `ngx_stream_core_content_phase()` 却非常简单,代码摘要如下:

```
cscf = ngx_stream_get_module_srv_conf(          //获取核心模块的 srv 配置
    s, ngx_stream_core_module);

cscf->handler(s);                                //执行配置的 content handler
```

这是因为 TCP/UDP 协议传输的内容千变万化,没有固定的处理模式,只能交给模块自己去解决:模块里必须自行设置对读写事件的监控,开辟读写缓冲区,使用会话对象的 `ctx` 保存状态,定义 handler 处理连接上的读写事件收发数据。

16.5.7 过滤数据

在 Stream 机制里发送数据时可以直接使用 `c->send`,但更好的方法是使用过滤链表,也就是 `ngx_stream_top_filter`,让数据过滤后再发送给客户端或后端。

目前的 Stream 过滤链表里只有一个过滤器 `ngx_stream_write_filter`,它内部使用 `ctx` 保存了待发送的数据块链,减轻了模块自己维护缓冲的工作量。

16.5.8 结束会话

函数 `ngx_stream_finalize_session()` 是 Stream 处理引擎的出口函数,可以无条件地跳出引擎处理循环,记录日志最后关闭连接,代码摘要如下:

```
void
ngx_stream_finalize_session(ngx_stream_session_t *s, ngx_uint_t rc)
{
    s->status = rc;                                //设置状态码,暂无意义

    ngx_stream_log_session(s);                    //调用 log 模块

    ngx_stream_close_connection(s->connection);  //关闭连接
}

static void
ngx_stream_log_session(ngx_stream_session_t *s)
{
    cmcf = ngx_stream_get_module_main_conf(s, ngx_stream_core_module);

    log_handler = cmcf->phases[NGX_STREAM_LOG_PHASE].handlers.elts;
    n = cmcf->phases[NGX_STREAM_LOG_PHASE].handlers.nelts;
```



```
for (i = 0; i < n; i++) {  
    log_handler[i](s);  
}  
}
```

//遍历 phases 数组的 log 模块
//执行模块的处理函数

结束会话的流程如图 16-12 所示。

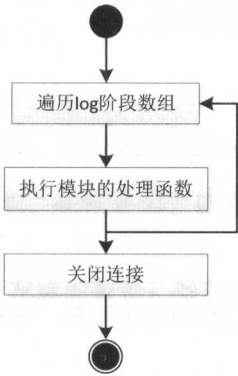


图 16-12 结束会话的流程

16.6 开发 stream 模块

本节将结合 Nginx 事件机制，使用 C++开发 stream 模块，实现几个最基本的 TCP 应用协议，包括：discard (RFC863)、time (RFC868) 和 chargen (RFC864)。

16.6.1 C++封装

事件机制和 Stream 机制相关的结构体较多，为了节约篇幅，本书不再列出 C++封装类的详细实现代码，仅做一个简略的介绍，具体的代码可参考 GitHub 资源。

这些 C++类都定义在名字空间 ngx::stream 里，不会与 HTTP 机制发生冲突。

NgxEvent

NgxEvent 封装 ngx_event_t，表示 Nginx 事件机制里的事件对象，成员函数 handler() 设置事件的处理函数，process() 执行处理函数，ready() 检查事件是否就绪，expired() 检查事件是否过期。

因为读写事件的操作有所不同，所以 NgxEvent 派生出两个子类 NgxReadEvent 和 NgxWriteEvent，用 wait_for() 函数直接添加对事件的监控并设置超时时间。

NgxConnection

NgxConnection 封装 ngx_connection_t，可以用 recv()/send() 收发数据。

NgxStreamCoreModule

NgxStreamCoreModule 代理 Stream 机制的核心模块 ngx_stream_core_module，成员函数 handler() 简化了模块注册处理函数的工作。

NgxStreamSession

NgxStreamSession 封装 ngx_stream_session_t，重载了多个形式的构造函数，可以直接从连接对象或事件对象构造，成员函数 send() 调用 ngx_stream_top_filter 发送数据。

NgxFilter

NgxFilter<Tag>封装 Stream 机制里的过滤链表，用法与 HTTP 机制的 NgxFilter 相同。

16.6.2 discard 协议

discard 协议是最简单的一个 TCP 应用协议，它没有任何输出，持续地接收数据并丢弃之。

模块设计

模块的设计如下：

- 模块名是 ndg_stream_discard_module，工作在 content 阶段；
- 指令 ndg_discard_time_out 确定客户端的超时时间；
- 指令 ndg_stream_discard 向 Stream 机制注册处理函数。

业务逻辑

函数 NdgStreamDiscardHandler::handler() 实现模块的业务逻辑：

```
static void handler(ngx_stream_session_t *s)    //处理函数
{
    NgxConnection conn(s);                      //获取连接对象

    NgxWriteEvent wev = conn.write_event();     //获取写事件对象
    wev.handler(                                  //设置写事件的处理函数
        [](ngx_event_t *ev){});                 //lambda 表达式定义一个空函数

    NgxReadEvent rev = conn.read_event();       //获取读事件对象
    rev.handler(                                  //设置读事件的处理函数
```

```

        &this_type::discard_read_handler); //读取数据并丢弃

rev.process(); //立即执行处理函数
}

```

discard 协议虽然简单，但正好可以示范 Stream 机制的基本处理流程。

当 Nginx 处理流程进入模块的 content handler 时，连接上的读写事件都处于未监控状态，必须要由模块自己决定处理的方式。

对于 discard 协议来说，写事件是不关心的，所以设置为一个空函数，意味着即使 socket 上有可写事件也会被完全忽略，不做任何处理。

读事件是 discard 协议要处理的，所以要为它设置处理函数。因为连接上可能已经收到了一些数据，所以要在函数里用 ready() 检查并处理，最后必须调用 wait_for() 函数加入事件机制的监控，等待客户端接下来发送的数据，等待的时间则由配置确定。

处理函数

函数 discard_read_handler() 是模块实际的工作函数，它处理连接上的读事件，使用一个很小的缓冲区读取客户端发送的数据，但不做任何处理：

```

static void discard_read_handler(ngx_event_t *ev) //处理读事件，丢弃数据
try //try 捕获函数里的任何异常
{
    NgxReadEvent rev(ev); //获取读事件对象
    NgxConnection conn(ev); //获取连接对象

    NgxException::fail(rev.expired(), NGX_ETIMEDOUT); //检查是否过期
    NgxException::fail(conn.closed()); //检查连接是否意外关闭

    if(rev.ready()) //检查读事件是否就绪
    {
        std::array<u_char, 64> buf; //64 字节的缓冲区
        ssize_t n = 0; //记录读取的字节数

        for(;;) //持续读取数据，直至出错
        {
            n = conn.recv(buf.data(), buf.size()); //读取最多 64 字节
            if(n <= 0) //<=0 就是发生了错误
            { break; } //退出循环
        }

        NgxException::fail(n != NGX_AGAIN); //如果不是 again 就结束会话
    }
}

```

```

    } //不可读则需要等待

    NgxStreamSession s(ev); //获取会话对象
    auto& cf = this_module::conf().srv(s); //取模块的配置
    rev.wait_for(cf.timeout, true); //设置超时时间,等待下一次读事件
}
catch(const NgxException& e) //捕获 catch 函数里的异常
{
    NgxStreamSession s(ev); //获取会话对象
    s.close(); //结束会话
}

```

在事件处理函数里必须检查超时和连接的状态,如果超时或者连接有错误就不应该继续。

之后的代码比较简单,循环调用 `conn.recv()` 读取数据,直至发生“错误”。如果错误码是 `NGX_AGAIN` 那么并不是真正的错误,表明此次客户端发送的数据已经读取完毕,需要等待下次读事件就绪;如果是其他错误码,那么就抛出异常,转入 `catch` 代码块结束会话。

16.6.3 time 协议

time 协议是另一个简单的 TCP 应用协议,它不关心任何输入,只输出一个当前的时间戳。

模块设计

time 协议的设计比 `discard` 要简单:

- 模块名是 `ndg_stream_time_module`,工作在 `content` 阶段;
- 指令 `ndg_stream_time` 向 Stream 机制注册处理函数。

业务逻辑

成员函数 `handler()` 忽略读事件,注册连接上写事件的处理函数:

```

static void handler(ngx_stream_session_t *s) //处理函数
{
    NgxConnection conn(s); //获取连接对象

    NgxReadEvent rev = conn.read_event(); //获取读事件对象
    rev.handler(&this_type::block_read_handler); //忽略读事件

    NgxWriteEvent wev = conn.write_event(); //获取写事件对象
    wev.handler(&this_type::time_write_handler); //设置写事件的处理函数

    wev.process(); //写事件必定就绪,所以立即执行处理函数
}

```

处理函数

函数 `time_write_handler()` 检查超时，然后分配内存发送时间戳字符串：

```
static void time_write_handler(ngx_event_t *ev)    //处理写事件，发送时间戳
{
    NgxWriteEvent wev(ev);                        //获取写事件对象
    NgxConnection conn(ev);                      //获取连接对象

    NgxException::fail(wev.expired(), NGX_ETIMEDOUT); //检查是否过期
    NgxException::fail(conn.closed());             //检查连接是否意外关闭

    if(!wev.ready())                             //检查写事件是否就绪
    {
        wev.wait_for(100);                       //如果不可写就等待 100 毫秒
        return;
    }

    NgxPool pool(conn);                          //从连接对象获取内存池

    NgxBuf buf = pool.buffer(20);                 //分配一小块内存
    buf.printf("%T", ngx_time());                 //打印时间戳
    buf.finish();                                 //标记缓冲区是 eof

    NgxChainNode n = pool.chain();                //分配一个链表节点
    n.set(buf);                                  //缓冲区添加到链表
    n.finish();                                  //链表结束

    NgxStreamSession s(conn);                    //获取会话对象
    s.send(n);                                   //发送数据

    if(conn->buffered)                            //检查数据是否被缓冲了
    {
        wev.wait_for(100);                       //可能是因为 socket 暂不可写
        //设置超时时间，等待下一次写事件
        return;
    }

    s.close();                                    //发送成功，关闭会话
}
```

代码里要注意的是内存池的使用，必须从连接对象里获取内存池。

`ngx_connection_t` 里的字段 `buffered` 标记了数据发送的状态，如果是 `true` 就意味着有部分数据没有被发送出去，被暂存了起来，必须注册写事件，当下次可写时再发送。

16.6.4 chargen 协议

chargen 协议是一个略微复杂的 TCP 应用协议，它持续地向客户端发送测试字符串。

模块设计

模块的设计如下：

- 模块名是 `ndg_stream_chargen_module`，工作在 `content` 阶段；
- 指令 `ndg_stream_chargen_count` 确定发送的字符串次数；
- 在模块的 `ctx` 里记录当前发送的次数；
- 指令 `ndg_stream_chargen` 向 Stream 机制注册处理函数。

业务逻辑

成员函数 `handler()` 初始化环境数据，注册写事件的处理函数并执行：

```
static void handler(ngx_stream_session_t *s)           //处理函数
{
    NgxConnection conn(s);                             //获取连接对象
    NgxStreamSession session(s);                       //获取会话对象

    auto& cf = this_module::conf().srv(session);       //获取配置信息
    auto& ctx_data = this_module::data(session);       //创建环境数据

    NgxValue::init(ctx_data.max_count, cf.max_count); //初始化 ctx 里的数据
    NgxValue::init(ctx_data.count, cf.max_count);    //count==0 即发送完毕
    NgxValue::init(ctx_data.num, 2);                //一次最多发送两行字符

    NgxReadEvent rev = conn.read_event();             //获取读事件对象
    rev.handler(&this_type::block_read_handler);     //忽略读事件

    NgxWriteEvent wev = conn.write_event();           //获取写事件对象
    wev.handler(&this_type::chargen_write_handler);  //设置写事件的处理函数

    wev.process();                                     //写事件必定就绪，所以立即执行处理函数
}
```

处理函数

写事件处理函数 `chargen_write_handler()` 需要检查 `ctx`，如果没有发送完就继续发送，发送完毕就关闭会话（代码里省略了超时检查）：

```
static void chargen_write_handler(ngx_event_t *ev) //处理写事件，发送
```

```

{
    NgxConnection conn(ev);           //获取连接对象
    NgxStreamSession s(ev);           //获取会话对象

    auto& ctx_data = this_module::data(s);           //获取环境数据

    for(;;)                               //循环发送字符串
    {
        send_chars(s, ctx_data);           //发送一些数据

        if(conn->buffered)                 //检查数据是否被缓冲了
        {
            NgxWriteEvent(ev).wait_for(100);       //可能是因为 socket 暂不可写
            return;                               //设置超时时间,等待下一次写事件
        }

        if(ctx_data.count == 0)             //发送成功,且全部发送完毕
        {
            break; }                       //退出循环
    }
    s.close();                             //结束会话
}

```

函数 `send_chars()` 真正发送数据:

```

static void send_chars(ngx_stream_session_t* s, ctx_type& ctx_data)
{
    NgxPool pool(s);                     //内存池

    static ngx_str_t lf_str = ngx_string("\n");           //换行字符

    auto& count = ctx_data.count;           //获取 ctx 里的计数信息
    auto& max_count = ctx_data.max_count;
    auto& num = ctx_data.num;

    std::vector<ngx_chain_t*> chains;           //vector 存放链表节点

    for(int i = num; i > 0 && count > 0; --i, --count)    //发送 num 个字符串
    {
        auto str = char_buffer(max_count - count);       //获取一个字符串

        NgxBuf buf = pool.buffer();           //分配缓冲区
        buf.range(str);                       //字符串添加到缓冲区

        NgxChainNode node = pool.chain();       //分配链表节点
        node.set(buf);                         //缓冲区添加到链表
    }
}

```

```

chains.push_back(node); //链表节点加入 vector

ngx_buf_t lf = pool.buffer(); //分配一个缓冲区
lf.range(lf_str); //存放换行符
lf.finish(count == 0); //只有发送完毕才能置 eof

node = pool.chain(); //分配链表节点
node.set(lf); //换行符添加到链表

chains.push_back(node); //链表节点加入 vector
}

for(size_t i = 0; i < chains.size()-1; ++i) //把链表节点串联起来
{ chains[i]->next = chains[i+1]; }

ngx_stream_session_t(s).send( //发送链表
    chains.empty() ? nullptr : chains.front()); //如果无数据可以传空指针
}

```

因为 C++ 类 ngx_pool 重载了构造函数，所以我们可以正确地从会话对象里获取内存池。

代码里的一个小技巧是使用 vector 临时存放了创建的所有链表节点，创建完之后再逐个连接，避免了在循环里判断头节点、调整指针的麻烦操作。

16.7 总结

本章简要介绍了 Nginx 的 Stream 机制，它处理 TCP/UDP 协议的原始字节流，可以看作是 HTTP 机制的“简化版”，所以两者可以互相参考对比学习，加深理解。

Stream 机制里的模块是一类独立的模块，类型是 NGX_STREAM_MODULE ("STRM")，通常称为 stream 模块。

stream 模块使用的函数指针表是 ngx_stream_module_t，包含配置解析相关的回调函数，存储配置数据使用的结构体是 ngx_stream_conf_ctx_t，这两个结构体与 HTTP 机制的配置结构很类似，仅缺少 location 相关的字段。

Stream 机制为 TCP/UDP 协议的处理划分了多个处理阶段，提供与 HTTP 机制类似的处理引擎和过滤引擎，所以我们可以把开发 http 模块的经验较容易地应用到开发 stream 模块上来。但也需要留意两者之间的区别，Stream 机制里使用的是会话对象 ngx_stream_session_t，而且需要我们自己处理连接上的读写事件，编写回调函数，使用 ctx 保存处理的中间状态，开发起来较 http 模块要麻烦很多。

本章的最后实现了一些 Stream 机制的 C++ 封装类，利用这些 C++ 类实现了 discard、time、chargin 等简单的 TCP 应用协议，可作为读者的开发参考，定制自己的 TCP/UDP 功能模块。

第 17 章

Nginx HTTP 机制

本书的前半部分都是在讲 HTTP 框架，包括模块体系、结构定义、处理引擎、开发方式等，但主要是从用户的角度进行的研究，没有深入底层实现。本章将结合事件机制，在第 7 章的基础上使用源码加流程图详细阐述 Nginx 处理 HTTP 协议的工作原理。

出于简化叙述的目的，本章主要研究应用的最广泛的 HTTP 1.1 协议，暂不涉及 HTTPS 和 HTTP 2.0。

17.1 结构定义

第 7 章中我们已经研究了 HTTP 框架里的一些关键数据结构，例如 `ngx_http_request_t`、`ngx_http_phase_engine_t` 等，本节再介绍其他几个与运行机制相关的数据结构。

17.1.1 `ngx_http_state_e`

处理 HTTP 请求的过程很复杂，所以 Nginx 使用一个枚举类型 `ngx_http_state_e` 来标记当前请求所在的处理状态，存储在请求对象的 `r->http_state` 字段里。

`ngx_http_state_e` 的定义如下：

```
// 位于 http/ngx_http_request.h
typedef enum {
    NGX_HTTP_INITING_REQUEST_STATE = 0,           // 初始状态，不使用
    NGX_HTTP_READING_REQUEST_STATE,              // 刚创建请求对象，正在读取请求数据
    NGX_HTTP_PROCESS_REQUEST_STATE,              // 请求头解析完毕，准备处理请求

    NGX_HTTP_CONNECT_UPSTREAM_STATE,             // 正在连接后端 upstream
}
```

```

NGX_HTTP_WRITING_UPSTREAM_STATE,           //向后端 upstream 发送数据
NGX_HTTP_READING_UPSTREAM_STATE,           //从后端 upstream 接收数据

NGX_HTTP_WRITING_REQUEST_STATE,             //响应请求, 向客户端发送数据
NGX_HTTP_LINGERING_CLOSE_STATE,             //延迟关闭状态
NGX_HTTP_KEEPAIVE_STATE                     //长连接 keepalive
} ngx_http_state_e;

```

但目前 Nginx 仅使用了 `NGX_HTTP_READING_REQUEST_STATE`、`NGX_HTTP_PROCESS_REQUEST_STATE` 等少数枚举值, 并没有用到所有的状态。

17.1.2 ngx_http_connection_t

`ngx_http_connection_t` 结构用来管理 HTTP 连接相关的配置信息和缓冲区数据, 定义摘要如下:^①

```

// 位于 http/ngx_http_request.h
typedef struct {
    ngx_http_addr_conf_t*    addr_conf; //保存 server 的基本配置信息
    ngx_http_conf_ctx_t*    conf_ctx;  //server{}里的配置结构数组

    ngx_chain_t*             busy;       //正在使用的数据块链
    ngx_int_t                nbusy;      //数据块链长度

    ngx_chain_t*             free;       //可复用的数据块链
} ngx_http_connection_t;

```

结构体里的两个关键字段是 `addr_conf` 和 `conf_ctx`, 前者表示监听端口对应的 server 配置, 后者则是 `server{}` 里的配置结构数组, 保存了所有模块在此 server 的配置信息。

17.1.3 ngx_http_request_t

`ngx_http_request_t` 是 Nginx 处理 HTTP 请求的核心数据结构, 在之前的章节里我们曾经反复提及, 这里再列出与 HTTP 框架执行流程相关的部分成员:

```

// 定义在 http/ngx_http_request.h
struct ngx_http_request_s {
    void**                ctx;           //处理请求的环境数据
    void**                main_conf;     //main 层次的配置信息数组
    void**                srv_conf;      //server 层次的配置信息数组
    void**                loc_conf;     //location 层次的配置信息数组

```

^① 在 Nginx 1.11.11 之前, `busy` 和 `free` 字段的类型是 `ngx_buf_t**`, 即缓冲区指针的数组。

```

ngx_http_event_handler_pt    read_event_handler;    //读事件处理函数
ngx_http_event_handler_pt    write_event_handler;    //写事件处理函数

ngx_pool_t*                  pool;                    //请求使用的内存池
ngx_buf_t*                   header_in;               //读取数据的缓冲区

ngx_http_connection_t*       http_connection;         //连接相关的配置信息

unsigned                      count:16;               //引用计数
unsigned                      blocked:8;              //异步或线程的引用计数

unsigned                      http_state:4;            //请求的处理状态

ngx_int_t                    phase_handler;           //当前处理阶段的索引号
ngx_http_handler_pt          content_handler;         //内容处理函数指针

unsigned                      lingering_close:1;       //延迟关闭的标志位
unsigned                      discard_body:1;         //丢弃请求体的标志位
unsigned                      reading_body:1;         //读取请求体的标志位

...                          //其他成员
};

```

请求结构体里有两个特殊的函数指针 `read_event_handler` 和 `write_event_handler`，它们是 HTTP 框架与 Nginx 事件机制结合的关键，Nginx 会依据请求的处理状态随时变动这两个函数指针，当连接上有读写事件发生时 HTTP 处理引擎就调用这两个函数执行恰当的功能。

`read_event_handler` 和 `write_event_handler` 的函数原型是：

```
typedef void (*ngx_http_event_handler_pt)(ngx_http_request_t *r);
```

注意它是处理 HTTP 相关读写事件的专用函数，入口参数不是事件对象 `ngx_event_t`，而是请求对象 `ngx_http_request_t`。

17.2 初始化连接

与 Stream 框架类似，HTTP 框架也定义了一系列的结构体来把配置文件里的监听端口与事件机制关联起来，例如 `ngx_http_listen_opt_t`、`ngx_http_conf_addr_t` 等，它们的结构、工作方式与 Stream 机制类似，故这里不再做详细介绍，请读者参考第 16 章。

在开始监听端口后，Nginx 事件机制会处理 `accept` 事件，调用 `ngx_event_accept()`

接受客户端的连接，然后调用 `ls->handler`，也就是 `ngx_http_init_connection()`，进入 HTTP 框架开始处理。

因为 HTTP 协议的格式是已知的，所以 HTTP 机制的工作流程与 Stream 机制略有不同，它读取请求头后才创建请求对象，再进入处理引擎。

17.2.1 建立连接

函数 `ngx_http_init_connection()` 是 HTTP 处理机制的起点，此时 Nginx 已经与客户端建立了连接，`socket` 是可写的（即可以向客户端发送数据），但不一定是可读的（可能还没有数据发过来），所以它要做的就是关注 `socket` 上的读事件，设置读事件的处理函数。

与 Stream 机制相同，函数首先要检查监听端口地址，查找对应的 `server{}` 配置信息，最后存放在 `hc->conf_ctx` 里，代码摘要如下：

```
hc = ngx_palloc(                                //创建连接相关的配置信息
    c->pool, sizeof(ngx_http_connection_t));

c->data = hc;                                    //暂时存放在这里备用

port = c->listening->servers;                    //检查监听端口的地址数组

if (port->naddrs > 1) {                          //监听端口对应多个地址
    ...                                          //在数组里找到正确的地址
} else {                                         //监听端口只对应一个地址
    addr = port->addrs;                         //地址数组里肯定只有一个元素
    hc->addr_conf = &addr[0].conf;            //取对应的配置信息
}

hc->conf_ctx =                                  //取 server 里的配置结构数组
    hc->addr_conf->default_server->ctx;         //关键字段是 default_server
```

确定了 `server` 之后，Nginx 就准备接收客户端发来的 HTTP 请求数据，为此需要在事件机制里监控 `socket` 的读事件，设置函数为 `ngx_http_wait_request_handler`。因为在接收请求头数据之前不可能发送 HTTP 响应数据，所以写事件不处理，设置为空函数 `ngx_http_empty_handler`：

```
rev = c->read;                                    //连接的读事件
rev->handler = ngx_http_wait_request_handler;    //可读时的处理函数
c->write->handler = ngx_http_empty_handler;      //暂时忽略写事件
```

如果监听端口配置了“`deferred`”参数，那么建立连接时 `socket` 上就已经有了可读的数据，就可以立即调用 `ngx_http_wait_request_handler()` 处理，否则就要设置超时时

间，并使用函数 ngx_handle_read_event() 把读事件加入 epoll 的监控，等待客户端发送数据，再次进入 HTTP 处理机制：

```
if (rev->ready) {                                     //读事件 ready，有数据可读
    rev->handler(rev);                                 //立即处理读事件，读取数据
    return;
}

ngx_add_timer(                                         //暂无数据可读，需等待
    rev, c->listening->post_accept_timeout);          //超时时间由配置确定

if (ngx_handle_read_event(rev, 0) != NGX_OK) {        //读事件加入 epoll 监控
    ngx_http_close_connection(c);                     //错误处理，监控失败需要关闭连接
    return;
}
```

函数 ngx_http_init_connection() 的流程图如图 17-1 所示。

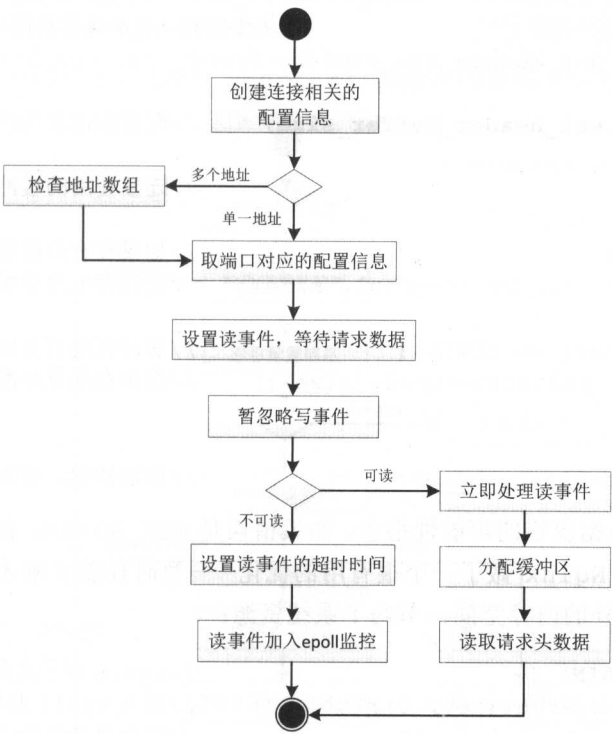


图 17-1 函数 ngx_http_init_connection() 的工作流程

17.2.2 等待数据

连接建立之后，一旦有数据可读 Nginx 就会调用函数 ngx_http_wait_request_

handler (即读事件关联的处理函数)。顾名思义,它只是“等待”客户端发送的请求数据,仅会被执行一次,为连接分配缓冲区,创建请求对象,之后调用 `ngx_http_process_request_line()` 解析请求行。

函数 `ngx_http_wait_request_handler()` 首先检查读事件是否超时,如果在限定时间里还没有收到客户端发来的数据就关闭连接:

```
c = rev->data;                                //从事件的 data 成员获得连接对象

if (rev->timedout) {                            //首先检查事件超时,即客户端未发送数据
    ngx_http_close_connection(c);              //超时没有发送数据,关闭连接
    return;
}
```

之后它从 `c->data` 里获取刚才在 `ngx_http_init_connection()` 里存放的 server 配置信息,分配读取数据使用的缓冲区,再调用 `c->recv` 接收数据:

```
hc = c->data;                                //从连接的 data 成员获得 server 配置信息
cscf = ngx_http_get_module_srv_conf(hc->conf_ctx, ngx_http_core_module);

size = cscf->client_header_buffer_size;      //配置的请求头缓冲区大小

b = c->buffer;                              //连接使用的缓冲区

if (b == NULL) {                             //如果没有分配缓冲区
    b = ngx_create_temp_buf(c->pool, size);    //在内存池里分配一块内存
    c->buffer = b;
} else if (b->start == NULL) {                //缓冲区没有关联实际的内存
    b->start = ngx_palloc(c->pool, size);      //在内存池里分配一块内存
}

n = c->recv(c, b->last, size);                //读取数据,存入缓冲区
```

`c->recv` 接收数据出错的可能性很大,如果错误是 `NGX_AGAIN`,表示客户端还没有发送来真正的数据,这里 Nginx 做了一个很有用的优化——暂时释放了刚才分配的内存,不会为无数据的连接保留额外的内存空间,节约了系统资源:

```
if (n == NGX_AGAIN) {                        //again 表示还没有数据可读
    ngx_handle_read_event(rev, 0);            //加入 epoll 监控,再次等待
    ngx_pfree(c->pool, b->start);              //释放缓冲区的内存,降低资源消耗
    b->start = NULL;                           //缓冲区指针清零,作为未分配的标记
    return;
}

if (n == NGX_ERROR) {                        //读取发生错误
```

```
ngx_http_close_connection(c); //需要立即关闭连接
return;
}
if (n == 0) { //读取 0 字节，客户端主动关闭了连接
    ngx_http_close_connection(c); //需要立即关闭连接
    return;
}
```

如果读取成功，那么缓冲区里就存放有客户端发来的 HTTP 请求数据，ngx_http_wait_request_handler() 的任务就完成了，所以就创建请求对象 ngx_http_request_t，把读事件的处理函数改为 ngx_http_process_request_line()，从解析请求行开始正式处理请求：

```
b->last += n; //此时成功读取了 n 个字节
c->data = ngx_http_create_request(c); //创建请求对象，正式开始处理

rev->handler = ngx_http_process_request_line; //调整读事件的处理函数
ngx_http_process_request_line(rev); //开始解析请求行数据
```

函数 ngx_http_wait_request_handler() 的流程图如图 17-2 所示。

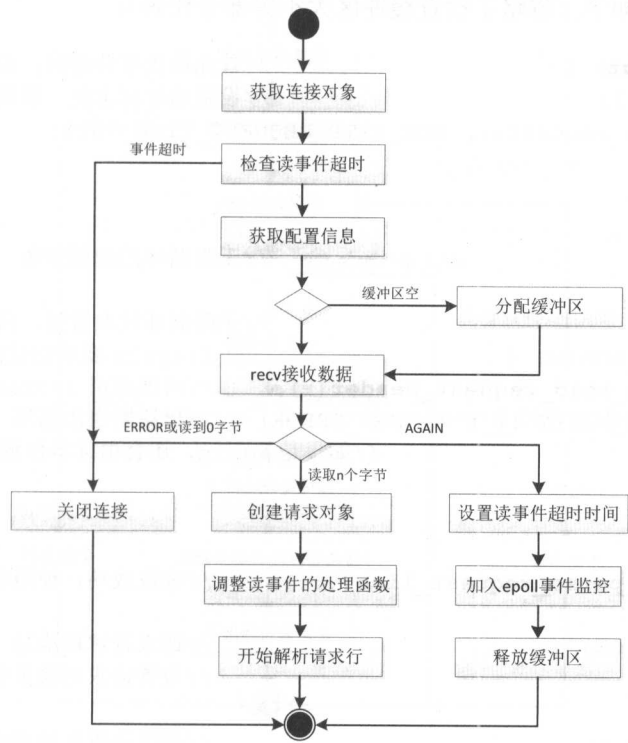


图 17-2 函数 ngx_http_wait_request_handler() 的工作流程

17.2.3 读取请求头

Nginx 使用函数 `ngx_http_process_request_line()` 和 `ngx_http_process_request_headers()` 来读取并解析 HTTP 请求行和请求头数据，这两个函数紧随着 `ngx_http_wait_request_handler()` 之后执行，由读事件直接触发。

请求行

函数 `ngx_http_process_request_line()` 使用无限 `for` 循环持续地从连接上读取数据，尝试获取 HTTP 请求行，工作流程的要点是：

- 读事件超时，直接结束请求；
- 读取发生错误，直接结束请求；
- 读不到数据（`NGX_AGAIN`），那么加入 `epoll` 监控，等待下一次读事件；
- 请求行解析出错，直接结束请求；
- 请求行不完整，那么加入 `epoll` 监控，等待下一次读事件；
- 请求行读取完毕，进入请求头处理函数继续处理。

函数的代码摘要如下（省略了检查缓冲区大小等部分代码）：

```
if (rev->timedout) {                                //首先检查事件超时，即客户端未发送数据
    c->timedout = 1;                                  //设置超时标志位，然后结束请求关闭连接
    ngx_http_close_request(r, NGX_HTTP_REQUEST_TIME_OUT);
    return;
}

rc = NGX_AGAIN;                                     //无限循环的起始参数

for ( ;; ) {                                         //无限循环读取数据，直至出错或读取成功
    if (rc == NGX_AGAIN) {                           //again 要求尝试读取数据
        n = ngx_http_read_request_header(r);         //内部调用 c->recv 读取数据
        if (n == NGX_AGAIN || n == NGX_ERROR) {      //出错则退出循环
            return;                                   //如果是 again，还会由读事件触发再次进入函数处理
        }
    }

    rc = ngx_http_parse_request_line(...);           //读取成功，使用状态机解析请求行

    if (rc == NGX_OK) {                               //请求行解析成功
        ...                                           //设置请求对象里的各个字段

        rev->handler =                                //修改读事件处理函数
            ngx_http_process_request_headers;         //改为处理请求头的函数
    }
}
```

```

ngx_http_process_request_headers(rev); //立即处理请求头数据

return;
}

if (rc != NGX_AGAIN) { //请求行解析错误, 结束请求
    ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
    return;
}

//无限 for 循环结束

```

代码里的关键是解析请求行的函数 `ngx_http_parse_request_line()`，它使用状态机高效地解析收到的数据。如果返回 `NGX_OK`，意味着 Nginx 已经完整接收到了客户端发来的请求行数据，可以继续处理后续的请求头；如果返回 `NGX_AGAIN`，则表示请求行数据不完整，需要由事件机制触发 `c->recv` 再接收更多的数据；其他的错误码则是解析错误，必须关闭连接结束请求。

函数的流程图如图 17-3 所示。

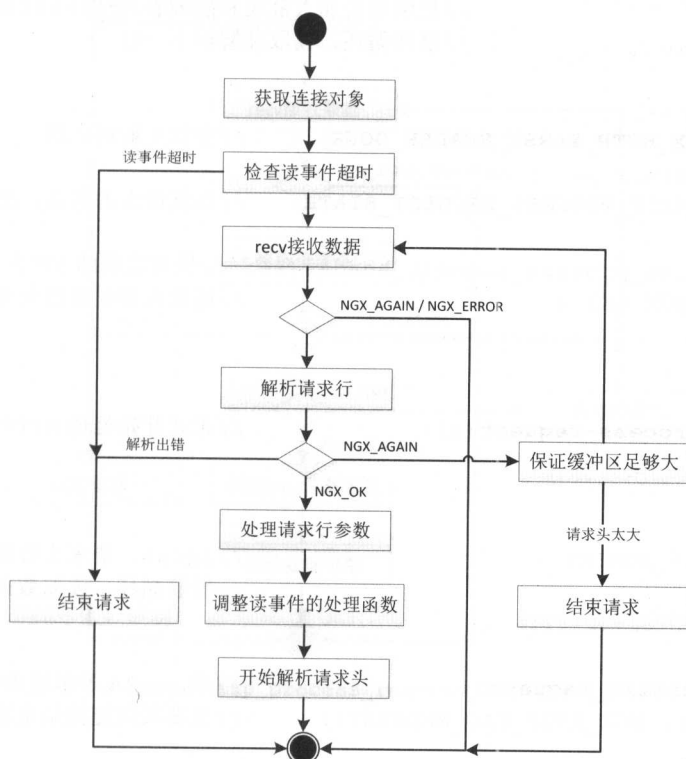


图 17-3 函数 `ngx_http_process_request_line()` 的工作流程

请求头

函数 `ngx_http_process_request_headers()` 的处理逻辑与 `ngx_http_process_request_line()` 类似, 同样使用无限 `for` 循环从连接上读取数据并解析:

```
rc = NGX_AGAIN; //无限循环的起始参数

for ( ;; ) {
    if (rc == NGX_AGAIN) { //again 要求尝试读取数据
        n = ngx_http_read_request_header(r); //内部调用 c->recv 读取数据
        if (n == NGX_AGAIN || n == NGX_ERROR) { //出错则退出循环
            return; //如果是 again, 还会由读事件触发再次进入函数处理
        }
    }

    rc = ngx_http_parse_header_line(r, //读取成功, 使用状态机解析请求头
        r->header_in, cscf->underscores_in_headers);

    if (rc == NGX_OK) { //成功解析完一行
        ... //把请求头加入链表和散列表, 使用 host 确定匹配的 server
        continue; //继续循环, 读取并解析下一行
    }

    if (rc == NGX_HTTP_PARSE_HEADER_DONE) { //全部头解析完毕
        r->http_state =
            NGX_HTTP_PROCESS_REQUEST_STATE; //设置请求的状态, 准备处理请求

        rc = ngx_http_process_request_header(r); //检查收到的 HTTP 请求头
        if (rc != NGX_OK) { //请求头错误则结束请求
            return;
        }

        ngx_http_process_request(r); //正式开始处理 HTTP 请求
        return;
    }

    if (rc == NGX_AGAIN) { //agian, 请求头数据不完整
        continue; //继续循环, 读取数据, 保证解析成功
    }

    ngx_http_finalize_request( //非 again 表示请求头解析出错
        r, NGX_HTTP_BAD_REQUEST); //需要关闭连接结束请求
    return;

} //无限 for 循环结束
```

这段代码里的关键是函数 `ngx_http_parse_header_line()`，它返回 `NGX_HTTP_PARSE_HEADER_DONE` 就表示成功接收到了完整的 HTTP 请求头，之后就调用 `ngx_http_process_request()` 正式进入 HTTP 处理引擎，调用 `http` 模块处理请求。

在解析请求行时，如果发现了 `host` 字段，Nginx 就会调用函数 `ngx_http_set_virtual_server()`，在散列表里查找匹配的 `server`，然后修改 `r->srv_conf` 和 `r->loc_conf`，也就是设置请求 `host` 所在 `server{}` 块的配置信息，这样在后续的处理时模块就可以得到正确的配置。

函数的流程图如图 17-4 所示。

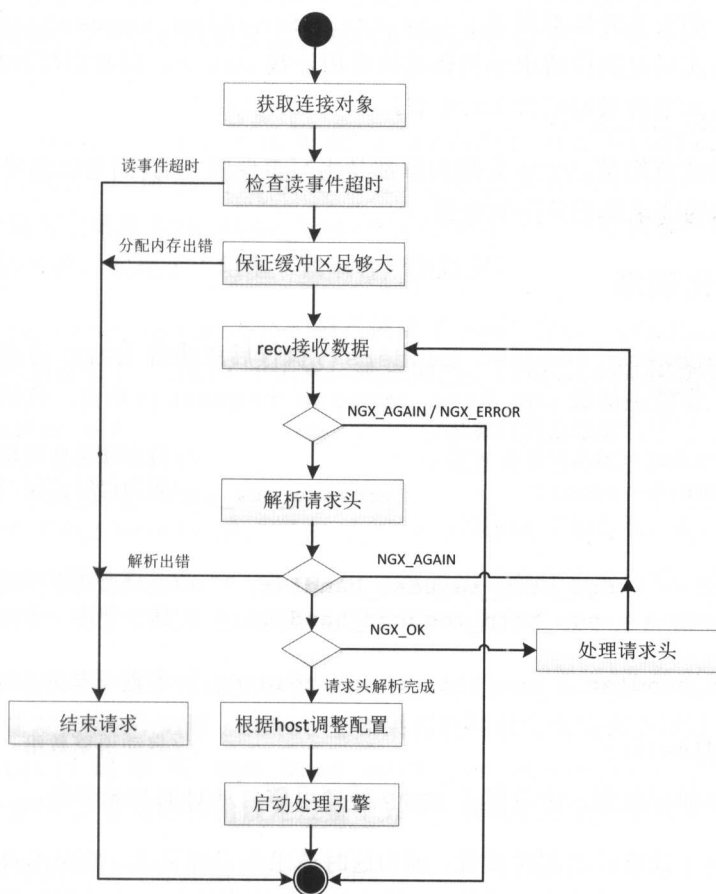


图 17-4 函数 `ngx_http_process_request_headers()` 的工作流程

17.3 执行引擎

在成功接收完 HTTP 请求头后, Nginx 就进入了 HTTP 处理引擎(相关的数据结构参见第 7 章), 在事件机制的驱动下由众多 http 模块联合处理客户端的请求, 最终产生合适的响应数据。

与 Stream 机制不同的是 HTTP 机制里有 location 的概念, 需要由请求 URI 来确定。Nginx 使用了二叉树来实现对 location 的快速查找定位, 但代码较繁琐, 故本节不予详细讨论, 而把精力集中在引擎每个阶段的处理逻辑上。

处理引擎运行的很多时候都调用了 `ngx_http_finalize_request()` 等函数, 用来“结束”请求, 但它其实只是操作请求结构体里的引用计数 `count`, 只有引用计数为 0 才会真正地结束请求, 阅读本节时最好结合 17.6 节。

还要提醒读者注意的是, HTTP 处理引擎都是由写事件触发的(因为已经读取完了请求头), 读事件由 17.4 节的请求体相关函数处理。

17.3.1 初始化引擎

`ngx_http_process_request_headers()` 执行后意味着 HTTP 请求的读事件初步处理完毕, 后续的工作就由函数 `ngx_http_process_request()` 负责, 代码摘要如下:

```
if (c->read->timer_set) {                                //检查读事件的超时设置
    ngx_del_timer(c->read);                               //因为已经读完, 所以不需要再检查
}

c->read->handler = ngx_http_request_handler;              //变动读写事件的处理函数
c->write->handler = ngx_http_request_handler;             //统一使用一个函数

r->read_event_handler = ngx_http_block_reading;          //设置请求的读处理函数, 屏蔽读

ngx_http_handler(r);                                     //启动引擎数组
```

代码不多, 但却很重要, 它设置了 HTTP 处理引擎启动时的初始参数。

函数首先检查了读事件的超时设置, 因为这时请求头已经读完, 暂时不再需要关注可读事件(除非再显式要求读取或丢弃请求体), 所以必须把读事件移出定时器红黑树, 避免触发不必要的超时事件。

接下来 Nginx 把连接上的读写事件处理函数都改为一个函数 `ngx_http_request_handler()`, 意味着之后连接上有任何事件发生都会由它来处理, 函数的代码是:

```
static void ngx_http_request_handler(ngx_event_t *ev)
{
    c = ev->data;                //从事件对象获取连接对象
    r = c->data;                  //再从连接对象获取请求对象

    if (ev->write) {              //如果是写事件
        r->write_event_handler(r); //调用请求里的写处理函数
    } else {                      //如果是读事件
        r->read_event_handler(r);  //调用请求里的读处理函数
    }
}
```

这样，`ngx_http_request_handler()`就把事件机制里的读写事件转换到了 HTTP 机制里的请求处理函数，在 HTTP 机制里无须关心事件机制的细节，只要专注于请求对象的处理就可以了。

`ngx_http_process_request()`里把 `r->read_event_handler` 设置为 `ngx_http_block_reading`，所以当连接上再触发可读事件时，`ngx_http_request_handler()`就会调用空函数 `ngx_http_block_reading()`，屏蔽读事件的处理，也就是不处理客户端发送的数据，实现了“阻塞”读取客户端数据。

`ngx_http_process_request()`的最后调用了 `ngx_http_handler()`，它启动了处理引擎，核心代码摘要如下（省略了子请求、延迟关闭、keepalive 等部分）：

```
r->phase_handler = 0;            //设置引擎数组起始序号，从头执行引擎数组
r->write_event_handler =        //设置请求的写处理函数，执行引擎数组
    ngx_http_core_run_phases;
ngx_http_core_run_phases(r);    //开始执行引擎数组，走 http 模块的处理
```

Nginx 先设置了请求的 `phase_handler`，它是引擎数组的索引号，指示了正在处理请求的当前模块，值为 0 表示要从头开始执行引擎数组，也就是说从 `POST_READ` 阶段开始处理。

最后 Nginx 设置了请求的写处理函数 `ngx_http_core_run_phases()`（参见 7.2.6 节），准备向客户端发送数据，后续 socket 上如果再有写事件发生都会经过函数 `ngx_http_request_handler()`调用到 `ngx_http_core_run_phases()`，使用 `r->phase_handler` 作为“游标”，让请求分阶段走过整个 http 模块职责链，最终完成请求的处理。

HTTP 处理引擎的启动过程如图 17-5 所示。

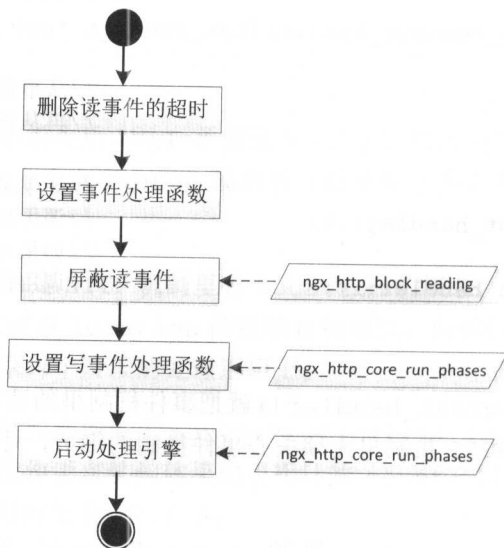


图 17-5 HTTP 处理引擎的启动过程

17.3.2 通用阶段

通用阶段处理函数 `ngx_http_core_generic_phase()` 管理的是 `POST_READ` 和 `PREACCESS` 两个阶段的模块，代码逻辑与 `Stream` 机制非常相似（参见 16.5.4 节）：

```

rc = ph->handler(r); //执行模块的处理函数

if (rc == NGX_OK) { //检查返回值，函数执行成功
    r->phase_handler = ph->next; //跳到下一个阶段
    return NGX_AGAIN; //要求引擎继续运行
}

if (rc == NGX_DECLINED) { //检查返回值，此模块拒绝处理
    r->phase_handler++; //使用下一个模块处理请求
    return NGX_AGAIN; //要求引擎继续运行
}

if (rc == NGX_AGAIN || rc == NGX_DONE) { //检查返回值，again 或 done
    return NGX_OK; //模块暂无法处理请求，等待下次可写事件，要求引擎暂时退出
}

ngx_http_finalize_request(r, rc); //发生错误，结束请求（减少引用计数）
  
```

从代码里可以知道 `POST_READ` 和 `PREACCESS` 两个阶段里模块处理函数返回值的含义：

■ `NGX_OK` : 处理成功，后续的本阶段模块均跳过，继续进入下一个阶段处理；

- NGX_DECLINED : 模块拒绝处理, 需要引擎里的下一个模块处理;
- NGX_AGAIN : 请求部分处理, 需要等待可写事件才能继续;
- NGX_DONE : 同 NGX_AGAIN 的含义;
- NGX_ERROR : 模块发生了严重错误, 引擎不需要再执行, 结束请求。

函数 ngx_http_core_generic_phase() 的流程图如图 17-6 所示。

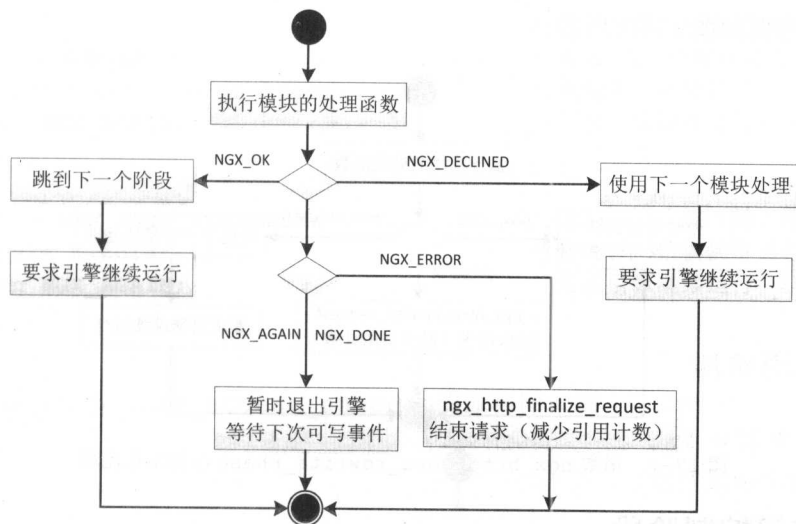


图 17-6 函数 ngx_http_core_generic_phase() 的工作流程

17.3.3 改写阶段

改写阶段处理函数 ngx_http_core_rewrite_phase() 管理 SERVER_REWRITE 和 REWRITE 两个阶段的模块, 这个阶段的模块工作是改写 URI, 所以逻辑比较简单:

```
rc = ph->handler(r); //执行模块的处理函数

if (rc == NGX_DECLINED) { //检查返回值, 模块正确运行
    r->phase_handler++; //使用下一个模块处理请求
    return NGX_AGAIN; //要求引擎继续运行
}

if (rc == NGX_DONE) { //检查返回值, done
    return NGX_OK; //模块暂无法处理请求, 等待下次可写事件, 要求引擎暂时退出
}

ngx_http_finalize_request(r, rc); //结束请求 (减少引用计数)
```

在 REWRITE 阶段执行的通常是 URI 的改写, 所以模块返回 NGX_DECLINED 表示模块正

确执行了操作, checker 会使用 `NGX_AGAIN` 让引擎继续运行, 如果模块返回的是 `NGX_OK` 或者其他 HTTP 状态码就表示 HTTP 请求已经处理完成, 将调用函数 `ngx_http_finalize_request()` 结束请求。

需要注意的是 `rewrite` 阶段模块执行成功应该返回 `NGX_DECLINED` 而不是 `NGX_OK`, 这样的原因是让同阶段的其他模块也有改写 URI 的机会, 所以 `rewrite` 模块不是互斥的。

函数的流程图如图 17-7 所示。

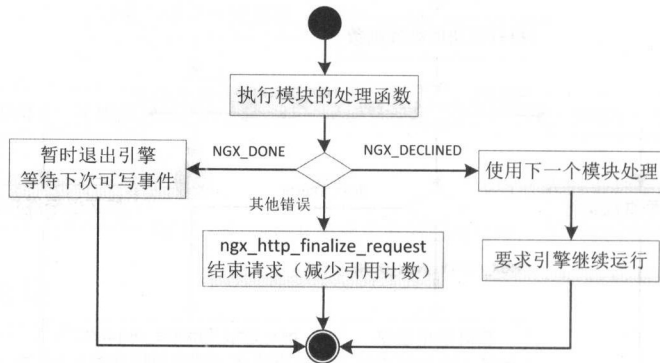


图 17-7 函数 `ngx_http_core_rewrite_phase()` 的工作流程

17.3.4 访问控制阶段

ACCESS 阶段使用的处理函数是 `ngx_http_core_access_phase()`, 它的代码逻辑与 `rewrite` 类似, 但多了访问控制相关的代码:

```

rc = ph->handler(r); //执行模块的处理函数

if (rc == NGX_DECLINED) { //检查返回值, 此模块拒绝处理
    r->phase_handler++; //使用下一个模块继续检查
    return NGX_AGAIN; //要求引擎继续运行
}

if (rc == NGX_AGAIN || rc == NGX_DONE) { //检查返回值, again 或 done
    return NGX_OK; //模块暂无法处理请求, 等待下次可写事件, 要求引擎暂时退出
}

... //访问控制权限检查代码, 见下

ngx_http_finalize_request(r, rc); //禁止访问, 结束请求 (减少引用计数)
  
```

Nginx 使用指令 “`satisfy all|any`” 来决定访问控制的条件, `all` 表示必须满足所

有条件，是“and”的关系；any 则正好相反，只要有一个条件满足就可以，是“or”的关系。对 access 模块的处理也就对这两种设置有区分，代码摘要如下：

```
if (clcf->satisfy == NGX_HTTP_SATISFY_ALL) { //必须所有模块都检查通过
    if (rc == NGX_OK) { //返回 ok，此模块检查通过
        r->phase_handler++; //使用下一个模块继续检查
        return NGX_AGAIN; //要求引擎继续运行
    }
} else { //仅一个模块检查通过即可
    if (rc == NGX_OK) { //返回 ok，此模块检查通过
        r->phase_handler = ph->next; //跳到下一个阶段
        return NGX_AGAIN; //要求引擎继续运行
    }
}

if (rc == NGX_HTTP_FORBIDDEN | ...) { //返回 401、403 等，此模块检查不通过
    r->phase_handler++; //使用下一个模块继续检查
    return NGX_AGAIN; //要求引擎继续运行
}
```

函数 ngx_http_core_access_phase() 的流程图如图 17-8 所示。

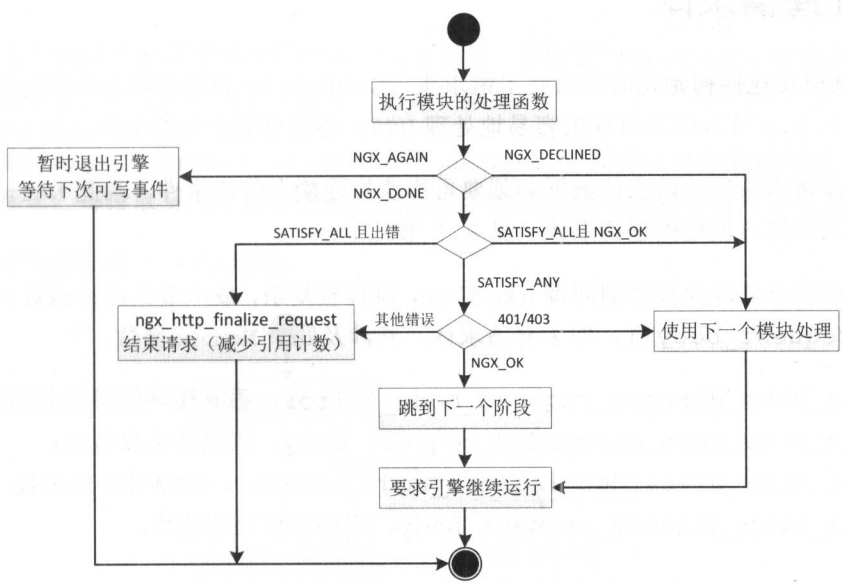


图 17-8 函数 ngx_http_core_access_phase() 的工作流程

17.3.5 内容产生阶段

函数 ngx_http_core_content_phase() 执行 content handler 产生具体的响应内

容, 代码已经在 7.2.6 节有所展示, 这里只给出流程图, 如图 17-9 所示。

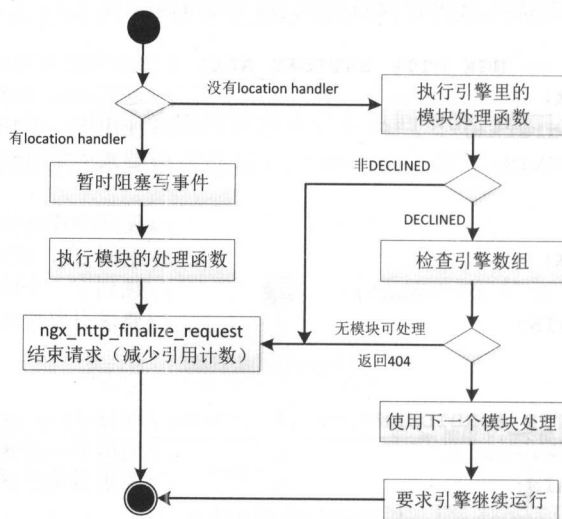


图 17-9 函数 ngx_http_core_content_phase() 的工作流程

17.4 处理请求体

HTTP 框架在连接初始化时读取完了请求头, 之后的 HTTP 处理引擎只关注写事件, “阻塞”了读事件(17.3.1 节), 所以可以很容易地处理 HTTP 协议里最常见的请求方法 GET 和 HEAD。

想要操作请求头后的请求体数据必须要再次监控连接上的读事件, 为此 Nginx 提供了一些函数帮助我们更方便地处理请求体 (见 8.5 节)。

由于处理请求体涉及较多的回调 handler, 流程较复杂, 故本节只讨论函数 ngx_http_discard_request_body(), 即丢弃请求体, 共涉及四个 Nginx 函数: ^①

- ngx_http_discard_request_body_filter, 丢弃缓冲区里收到的数据;
- ngx_http_read_discarded_request_body, 读取并丢弃数据;
- ngx_http_discarded_request_body_handler, 读事件处理函数;
- ngx_http_discard_request_body, 启动丢弃处理流程。

^① HTTP 协议里的请求体有确定长度和分块传输 (chunked) 两种形式, 简单起见本书只研究确定长度相关的代码, 便于理解 Nginx 的工作原理。

17.4.1 丢弃缓冲区数据

函数 ngx_http_discard_request_body_filter() 简单地检查缓冲区, 不做任何处理, 只调整指针清空缓冲区 (即丢弃数据), 如果长度已经达到 content_length_n 就表明 body 数据已经读取完毕:

```
static ngx_int_t
ngx_http_discard_request_body_filter(ngx_http_request_t *r, ngx_buf_t *b)
{
    size_t size = b->last - b->pos; // 获取缓冲区里的数据长度

    if ((off_t) size > r->headers_in.content_length_n) { // 超过了预定的长度
        b->pos += (size_t) r->headers_in.content_length_n; // 只移动预定的长度
        r->headers_in.content_length_n = 0; // 长度置为 0, 读取完毕
    } else { // 读取了较少的数据
        b->pos = b->last; // 移动指针, 清空缓冲区
        r->headers_in.content_length_n -= size; // 减少长度
    }

    return NGX_OK; // 丢弃完毕, 返回 OK
}
```

代码里的关键操作是调整缓冲区的 b->pos 指针, 直接置为 b->last, 也就是消费数据, 但并没有对数据做任何处理, 从而实现了“丢弃”数据。

请求对象里的 r->headers_in.content_length_n 是请求体数据的长度, 它同时也作为读取的标志, 每读取 size 长度的数据就减少, 当减少到 0 就意味着所有数据读取完毕。

函数的流程图如图 17-10 所示。

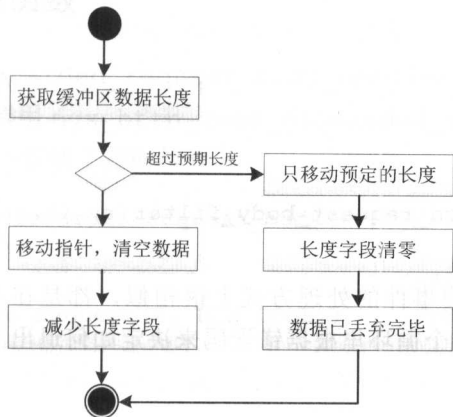


图 17-10 函数 ngx_http_discard_request_body_filter() 的工作流程

17.4.2 读取并丢弃数据

当连接可读时, Nginx 会调用函数 `ngx_http_read_discarded_request_body()`, 使用一个 4K 大小的缓冲区持续地读取客户端发来的数据, 然后用 `ngx_http_discard_request_body_filter()` 检查并丢弃缓冲区里的数据, 代码摘要如下:

```
u_char  buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];           //4K 大小的缓冲区

for ( ;; ) {                                             //无限循环接收数据
    if (r->headers_in.content_length_n == 0) {          //检查长度, 0 表示读取完
        r->read_event_handler = ngx_http_block_reading; //调整读事件函数, 不再读取
        return NGX_OK;                                   //已经成功丢弃了所有数据
    }

    if (!r->connection->read->ready) {                   //连接暂不可读
        return NGX_AGAIN;                                //等待下次可读事件
    }

    size = (size_t) ngx_min(...);                       //确定要读取的数据长度

    n = r->connection->recv(                               //从连接上读取数据
        r->connection, buffer, size);

    if (n == NGX_ERROR) {                                 //读取出错
        r->connection->error = 1;                          //设置错误标志位
        return NGX_OK;                                    //因为是丢弃所以不算错误
    }

    if (n == NGX_AGAIN) {                                 //连接暂不可读
        return NGX_AGAIN;                                //等待下次可读事件
    }

    if (n == 0) {                                         //客户端关闭了连接
        return NGX_OK;                                    //因为是丢弃所以不算错误
    }

    rc = ngx_http_discard_request_body_filter(r, &b);    //检查并丢弃缓冲区里的数据
}                                                         //无限 for 循环结束
```

可以看到, Nginx 在读事件的处理方式上很相似, 都是在一个无限 `for` 循环里调用 `c->recv` 接收数据, 并在这个循环里根据错误码来决定如何退出函数。

函数 ngx_http_read_discarded_request_body() 通常只有 NGX_AGAIN 和 NGX_OK 两种返回值（这里暂忽略 chunked）。如果连接不可读那么就返回 NGX_AGAIN；如果 r->headers_in.content_length_n == 0，就表示请求体数据正常读取完毕，函数就返回 NGX_OK。其他读取发生错误的情况，也可以认为是丢弃请求体成功，所以同样返回 NGX_OK。

函数的流程图如图 17-11 所示。

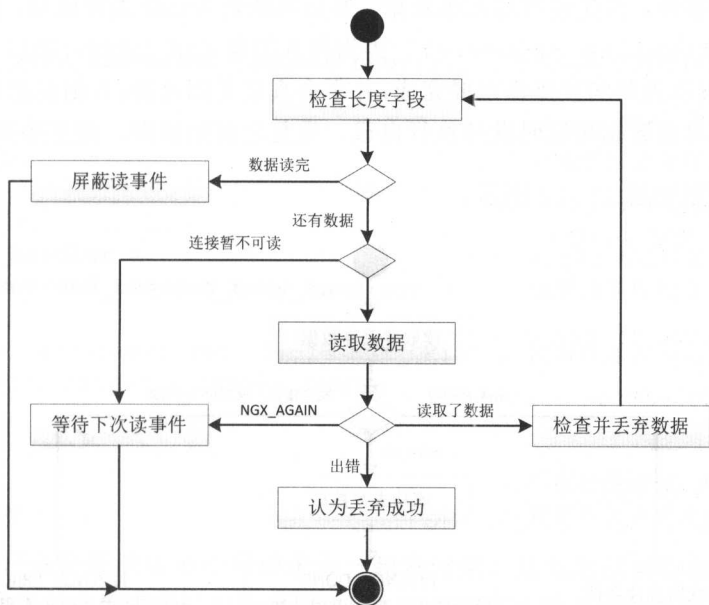


图 17-11 函数 ngx_http_read_discarded_request_body() 的工作流程

17.4.3 读事件处理函数

函数 ngx_http_discarded_request_body_handler() 是 Nginx 在丢弃请求体时使用的读事件处理函数，调用 ngx_http_read_discarded_request_body()，并检查它的返回值，代码摘要如下（省略了超时检查）：

```
rc = ngx_http_read_discarded_request_body(r); //读取并丢弃缓冲区里的数据

if (rc == NGX_OK) { //已经成功丢弃了请求体
    ngx_http_finalize_request(r, NGX_DONE); //传入 done，减少引用计数
    return;
}

if (rc >= NGX_HTTP_SPECIAL_RESPONSE) { //读取 chunked 发生错误
```

```

    ngx_http_finalize_request(r, NGX_ERROR);    //结束请求
    return;
}

if (ngx_handle_read_event(rev, 0) != NGX_OK) { //还需读取数据, 监控读事件
    ngx_http_finalize_request(r, NGX_ERROR);    //加入 epoll 失败则结束请求
    return;
}

```

函数处理读事件, 首先读取请求体数据, 然后判断是否已经丢弃成功, 如果丢弃成功就调用 `ngx_http_finalize_request()`, 因为传入的是 `NGX_DONE`, 所以只是减少请求对象里的引用计数, 表示关联的读操作已经完成, 并不会真正关闭连接; 否则就把读事件加入 `epoll` 监控, 当连接上再有数据可读时继续执行自己, 重复之前的步骤, 直至数据完全丢弃。

函数的流程图如图 17-12 所示。

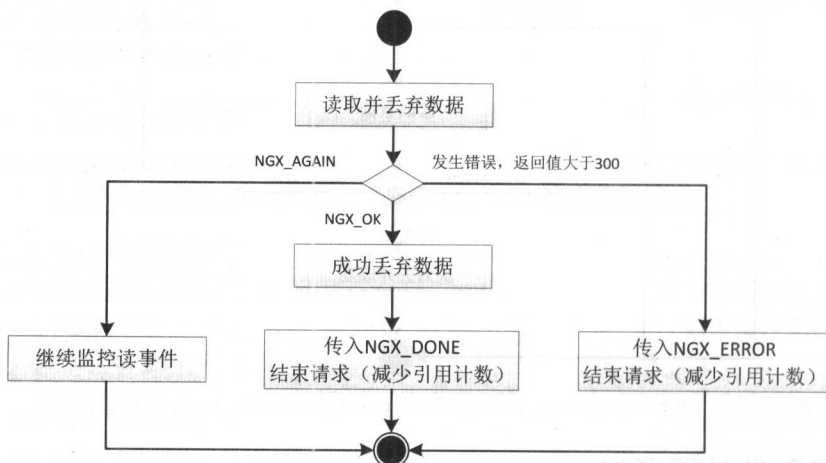


图 17-12 函数 `ngx_http_discarded_request_body_handler()` 的工作流程

17.4.4 启动丢弃处理

`ngx_http_discard_request_body()` 是 Nginx 对外提供的丢弃请求体的接口, 也是我们编写 `http` 模块时经常使用的函数, 它综合使用了之前的三个函数, 代码摘要如下 (省略了子请求和分块相关代码):

```

rev = r->connection->read;                                //从连接对象获取读事件

if (r->headers_in.content_length_n <= 0) {                 //如果无请求体数据则无须丢弃
    return NGX_OK;                                          //直接返回丢弃成功
}

```

```

size = r->header_in->last - r->header_in->pos; //检查缓冲区里收到的数据

if (size) { //如果缓冲区里有数据就需要丢弃
    rc = ngx_http_discard_request_body_filter(r, r->header_in);

    if (r->headers_in.content_length_n == 0) { //丢弃成功则无须再读取数据
        return NGX_OK; //直接返回丢弃成功
    }
}

rc = ngx_http_read_discarded_request_body(r); //读取并丢弃缓冲区里的数据

if (rc == NGX_OK) { //丢弃成功则无须再读取数据
    return NGX_OK; //直接返回丢弃成功
}

r->read_event_handler = //again 表示还需要读取后续数据
    ngx_http_discarded_request_body_handler; //设置读事件处理函数

if (ngx_handle_read_event(rev, 0) != NGX_OK) { //读事件加入 epoll 监控
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}

r->count++; //引用计数增加, 表示有操作未完成
r->discard_body = 1; //置正在丢弃请求体的标志位

```

很多情况下客户端会把请求体连同请求头一起发过来, 这些数据就存放在缓冲区 `r->header_in` 里, 在解析完请求头时仍然未被使用, 所以需要先检查这部分数据, 调用 `ngx_http_discard_request_body_filter()` 函数丢弃, 如果还有剩余的请求体, 那么就需
要监控读事件继续读取数据, 都交给 `ngx_http_discarded_request_body_handler()` 处理。

函数的流程图如图 17-13 所示。

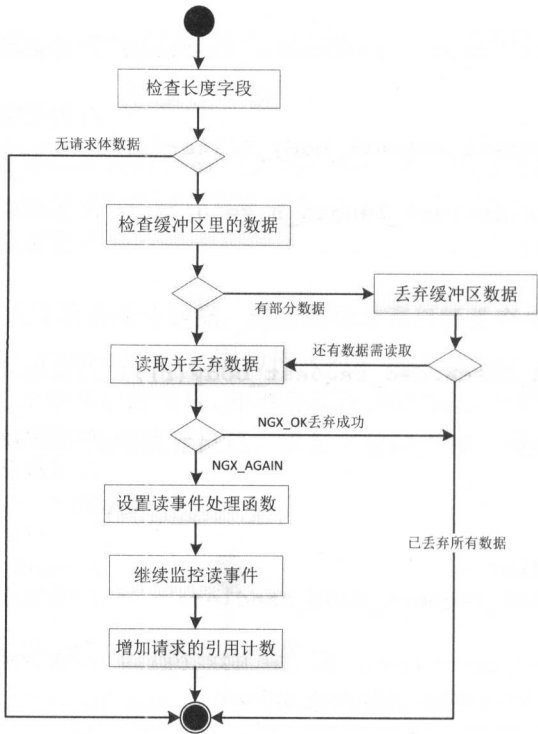


图 17-13 函数 ngx_http_discard_request_body() 的工作流程

17.5 发送数据

CONTENT 阶段是 HTTP 处理引擎的最后一个处理阶段（LOG 阶段只记录日志，不处理请求），Nginx 会使用过滤引擎（见 7.3 节）发送响应数据，这时将再次监控连接上的写事件，但关注的重点是可写，持续地发送被缓冲的数据。

Nginx 利用定时器实现了限速功能，写事件的标志位 wev->delayed 标记了是否限速，r->limit_rate 和 r->limit_rate_after 是限速的条件，处理逻辑主要集中在 ngx_http_write_filter_module 里，本书暂不研究。

17.5.1 发送初始化

函数 ngx_http_set_write_handler() 设置了发生响应数据时的写事件处理函数，代码摘要如下：

```
r->http_state = NGX_HTTP_WRITING_REQUEST_STATE; //当前的状态是正在发送数据
```

```

r->read_event_handler = r->discard_body ?           //设置读事件处理函数
    ngx_http_discarded_request_body_handler:
    ngx_http_test_reading;

r->write_event_handler = ngx_http_writer;           //设置写事件处理函数

wev = r->connection->write;                         //获取连接的写事件

if (wev->ready && wev->delayed) {                   //处于限速状态则暂不写数据
    return NGX_OK;                                 //不加入 epoll 监控
}

if (!wev->delayed) {                                //不限速
    ngx_add_timer(wev, clcf->send_timeout);         //设置写事件的超时时间
}

ngx_handle_write_event(wev, clcf->send_lowat);      //加入监控, 等待写事件

```

函数的代码比较简单, 关键是重新设置了请求的读写处理函数。

对于读事件, 由于此时已经开始向客户端发送响应数据, 就意味着客户端的请求数据已经读取完毕了, 所以不再关心读事件, 需要使用函数 `ngx_http_discarded_request_body_handler()` 丢弃请求体, 或者使用函数 `ngx_http_test_reading()` 检查客户端断连事件, 但无论是哪个函数, 都不会再实际处理客户端发送的数据。

对于写事件, 由于 HTTP 处理引擎已经执行完毕, 所以改为函数 `ngx_http_writer()`, 之后再发生可写事件就不会执行引擎函数 `ngx_http_core_run_phases()`。

如果不限速, Nginx 就把写事件加入 `epoll` 监控, 等待事件机制的触发, 调用 `ngx_http_writer()` 尽快地发送数据

17.5.2 事件处理函数

在 HTTP 请求处理的最后阶段使用的写事件处理函数是 `ngx_http_writer()`, 它调用过滤链表发送或缓存响应数据, 最后结束请求, 代码摘要如下 (省略了限速处理):

```

c = r->connection;                                //获取连接对象
wev = c->write;                                    //获取连接的写事件

if (wev->timedout) {                               //写事件超时
    c->timedout = 1;                                //设置超时标记, 结束请求
    ngx_http_finalize_request(r, NGX_HTTP_REQUEST_TIME_OUT);
    return;
}

```

```

}

rc = ngx_http_output_filter(r, NULL);           //使用过滤引擎发送数据

if (rc == NGX_ERROR) {                         //发送出错则结束请求
    ngx_http_finalize_request(r, rc);
    return;
}

if (r->buffered) {                             //成功，但数据被缓存暂未发送
    ngx_add_timer(wev, clcf->send_timeout);    //设置超时，等待下次写事件再发送
    ngx_handle_write_event(wev, clcf->send_lowat);
    return;
}

r->write_event_handler =                       //全部发送完毕
    ngx_http_request_empty_handler;           //阻塞写事件，不再发送数据

ngx_http_finalize_request(r, rc);             //结束请求

```

ngx_http_writer() 的代码不多，核心的发送和缓存数据工作都位于过滤链表函数 ngx_http_output_filter() 里，它只检查函数的返回值，如果 r->buffered 为 0，则表示所有响应数据都已经发送完毕，不再需要监控写事件，请求已经处理完毕，调用 ngx_http_finalize_request() 结束请求。

ngx_http_writer() 的流程图如图 17-14 所示。

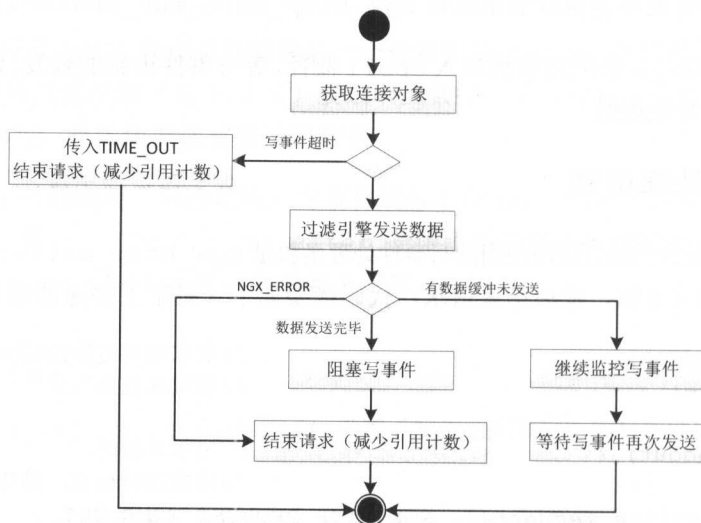


图 17-14 ngx_http_writer() 的工作流程

17.6 结束请求

在处理 HTTP 请求的过程中可能会因为各种各样的原因结束请求，既有正常的发送完数据结束，也有权限检查不通过而结束，还有其他的很多情况（超时、读取错误、缓冲区满、内部异常等）导致的非正常结束。Nginx 使用多个函数应用在不同的场合，其中最常用的就是 `ngx_http_finalize_request()`。

本节将详细分析这些重要函数的工作原理。

17.6.1 释放请求资源

HTTP 协议支持长连接，可以反复重用一个连接，所以在一个请求结束时需要记录访问日志，然后释放请求相关的资源，但并不真正地关闭连接（归还连接池）而是重用。

函数 `ngx_http_free_request()` 负责释放请求的内存等资源，代码摘要如下：

```
if (r->pool == NULL) {                //请求的内存池已经清空
    return;                            //即已经释放了资源，不需要后续操作
}

cln = r->cleanup;                      //请求自己的清理动作链表
while (cln) {                          //遍历清理动作链表
    if (cln->handler) {                 //执行请求相关的清理动作
        cln->handler(cln->data);
    }
    cln = cln->next;
}
ngx_http_log_request(r);              //执行 log 模块，记录日志

pool = r->pool;                        //请求使用的内存池
r->pool = NULL;                       //内存池指针清零
ngx_destroy_pool(pool);               //销毁请求的内存池
```

HTTP 请求处理过程中使用到的资源主要就是内存，所以函数检查 `r->pool`，如果它是空指针，就表明资源已经释放了，否则就需要调用 `ngx_destroy_pool()` 销毁内存池，同时释放在这个内存池里分配的所有内存。所以，我们在编写 http 模块功能代码时应当尽量使用 `r->pool` 来分配内存，这样在结束请求时可以一次性归还系统，没有内存碎片也提高了效率。

HTTP 处理也使用了清理函数链表机制（参见 3.4.4 节），用来销毁与请求相关的其他资源，与内存池 `ngx_pool_t` 类似，功能上有些重复，通常情况下我们不会使用 HTTP 的清理函数，直接使用内存池清理机制更好。

读者还需要注意 Nginx 在这个函数里调用了 `ngx_http_log_request()`，执行了所有的 `log` 模块，记录日志或者其他的收尾工作。

17.6.2 检查引用计数结束请求

函数 `ngx_http_close_request()` 操作请求结构体里的引用计数 `r->count`，尝试“关闭”HTTP 请求。如果引用计数大于零，说明请求还有关联的其他操作（例如读取数据、访问后端服务、子请求等）没有完成，不能关闭，否则就调用 `ngx_http_free_request()` 和 `ngx_http_close_connection()` 释放请求相关的资源。

函数的代码摘要如下：

```
if (r->count == 0) {                                //检查引用计数
    ngx_log_error(NGX_LOG_ALERT, ...);              //至少为 1，表示有操作，否则就是错误
}

r->count--;                                          //减少引用计数

if (r->count || r->blocked) {                          //还有其他关联的操作未完成
    return;                                          //不能结束请求
}

ngx_http_free_request(r, rc);                        //释放请求的内存池等资源，记录日志
ngx_http_close_connection(c);                       //释放连接的内存池资源，归还连接对象
```

函数的流程图如图 17-15 所示。

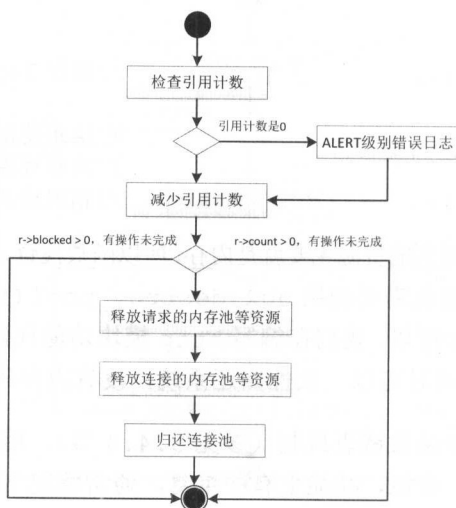


图 17-15 函数 `ngx_http_close_request()` 的工作流程

17.6.3 检查状态结束请求

函数 `ngx_http_finalize_connection()` 检查请求当前的处理状态，包括正在读取或丢弃请求体、延迟关闭、keepalive 等，然后调用 `ngx_http_close_request()` 减少引用计数结束请求，代码摘要如下：

```

if (r->main->count != 1) {                                //有多个相关操作尚未完成
    ngx_http_close_request(r, 0);                          //减少引用计数，尝试结束请求
    return;
}

if (!ngx_terminate                                        //进程不是结束运行状态
    && !ngx_exiting                                       //进程不是正在退出状态
    && r->keepalive                                         //请求是 keepalive
    && clcf->keepalive_timeout > 0)                       //有 keepalive 的超时时间
{
    ngx_http_set_keepalive(r);                             //不关闭请求，而是设置 keepalive
    return;
}

if (clcf->lingering_close == ...)                          //要求延迟关闭
{
    ngx_http_set_lingering_close(r);                       //关闭写事件，设置读事件处理函数
    return;
}

ngx_http_close_request(r, 0);                             //减少引用计数，尝试结束请求

```

`ngx_http_finalize_connection()` 首先检查主请求引用计数，如果 `r->main->count` 大于 1，那么只减少引用计数，并不会真正结束请求。当 `r->main->count` 为 1 时意味着请求上的所有相关操作都已经完成，就可以真正地“结束”请求了。

因为 HTTP 协议规定了 keepalive、lingering_close 等特性，有这两种情况的请求还不能真正地关闭连接，所以 Nginx 分别使用 `ngx_http_set_keepalive()` 和 `ngx_http_set_lingering_close()` 两个函数来处理后续连接上的读写事件：前者清除连接上残余的数据，释放前一个请求的资源，再次监控读事件；后者关闭写事件，在限定时间内处理读事件，最后关闭连接。

`ngx_http_finalize_connection()` 的流程图如图 17-16 所示。

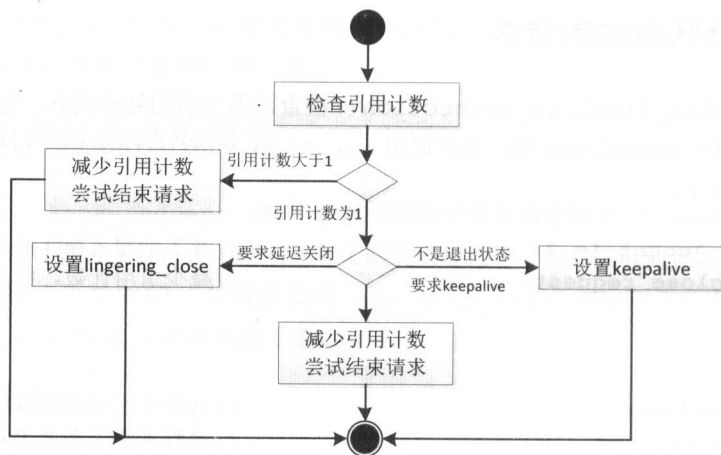


图 17-16 ngx_http_finalize_connection() 的工作流程

17.6.4 综合处理结束请求

ngx_http_finalize_request() 是 Nginx HTTP 机制里的关键函数，在执行引擎、收发数据时反复出现，每当一个请求相关的操作完成时都会调用此函数，作用非常重要。

ngx_http_finalize_request() 可以在入口里传递一个错误码，它依据这个错误码再调用 ngx_http_finalize_connection()、ngx_http_close_request() 等函数，以“适当”的方式“结束”请求——减少引用计数，或者真正地关闭连接。

ngx_http_finalize_request() 的代码摘要如下（省略了子请求等逻辑）：

```

if (rc == NGX_DONE) {                                     //有关联操作未完成，使用 done
    ngx_http_finalize_connection(r);                       //减少引用计数，尝试结束请求
    return;
}

if (rc == NGX_DECLINED) {                                 //模块拒绝处理请求
    r->content_handler = NULL;                             //清空内容处理函数指针
    r->write_event_handler = ngx_http_core_run_phases;     //重置写事件处理函数
    ngx_http_core_run_phases(r);                           //重走处理引擎再次处理
    return;
}

if (rc == NGX_ERROR||...)                                //发生了致命的错误
{
    ngx_http_terminate_request(r, rc);                     //减少引用计数，尝试结束请求
    return;
}
  
```

```

if (rc >= NGX_HTTP_SPECIAL_RESPONSE||...)           //标准的 HTTP 错误码
{
    c->read->handler = ngx_http_request_handler;      //重置读写事件处理函数
    c->write->handler = ngx_http_request_handler;

    ngx_http_finalize_request(r,                     //递归调用自己
        ngx_http_special_response_handler(r, rc));   //发送错误页面后再结束请求
    return;
}

if (r->buffered || ...) {                            //有数据未发送完
    ngx_http_set_write_handler(r);                   //设置写事件, 继续发送数据
    return;                                           //暂不结束请求, 但最后还会进入本函数
}

r->done = 1;                                          //done 标志位, 请求已经处理完毕
r->write_event_handler =                             //不再关注写事件, 即不再发送数据
    ngx_http_request_empty_handler;

ngx_del_timer(c->read);                              //删除读事件的超时设置
ngx_del_timer(c->write);                             //删除写事件的超时设置

if (c->read->eof) {                                   //客户端主动断连
    ngx_http_close_request(r, 0);                    //关闭连接, 释放资源
    return;
}

ngx_http_finalize_connection(r); //减少引用计数, keepalive 或 lingering_close

```

上面的代码已经对 Nginx 的源码做了大幅度的简化, 去掉了很多细节操作, 这样有助于我们更好地梳理 `ngx_http_finalize_request()` 的工作流程。

要想透彻地理解 `ngx_http_finalize_request()` 的工作原理必须要对整个 HTTP 框架有总体的了解, 因为框架里的很多地方——如处理引擎、读取数据、发送数据等流程都调用了它来“结束”请求, 而且它还会递归地调用自身, 流程的复杂度较高, 请读者结合图 17-17 流程图来加深认识。

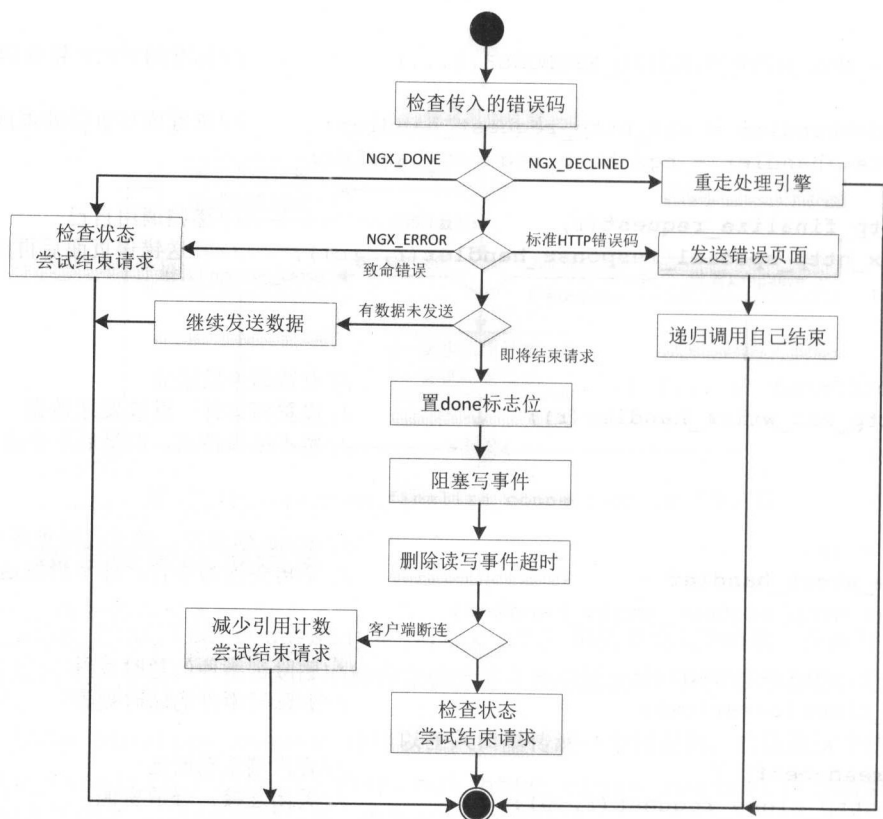


图 17-17 函数 ngx_http_finalize_request() 的工作流程

17.7 总结

本章简要研究了 Nginx HTTP 机制里的建立连接、处理引擎、收发数据和结束请求等流程。

虽然 HTTP 协议使用的也是 TCP，但协议的格式是已知的，所以 HTTP 机制的工作流程与 Stream 机制略有不同，关键点在于对 accept 连接后读写事件的处理。请求结构体里的两个函数指针：read_event_handler 和 write_event_handler，它们指向 HTTP 机制里真正的事件处理函数。

在建立连接后，HTTP 机制首先关注读事件，使用 ngx_http_wait_request_handler()、ngx_http_process_request_line() 和 ngx_http_process_request_headers() 这三个回调函数读取 HTTP 请求头。在事件机制的帮助下，处理没有任何阻塞，有数据就会立即读取并解析，没有数据或数据不完整则暂停处理，等待下次可读事件。

请求头读取完毕后, HTTP 机制统一使用函数 `ngx_http_request_handler()` 处理读写事件: 读事件暂时屏蔽, 而写事件则触发处理引擎。

处理引擎关注的是写事件, 在引擎里分成了 `access/rewrite/content` 等阶段, 使用 `checker` 调用 `http` 模块的 `handler` 处理请求, 然后检查模块的返回值, 决定引擎的下一步动作: 继续运行引擎、暂时退出引擎 (等待写事件) 或者出错结束请求的处理。

读取请求体关注的是读事件, 独立于处理引擎的写事件。本章仅分析了丢弃请求体的功能, Nginx 使用一个很小的缓冲区来接收客户端发送过来的数据, 使用请求结构体里的 `content_length_n` 作为计数器, 持续地接收并丢弃, 直至计数器减为 0。

处理引擎最后的运行结果是发送数据 (正常响应或错误反馈)。在数据发送阶段, HTTP 机制把 `write_event_handler` 指向了 `ngx_http_writer`, 不再运行处理引擎。`ngx_http_writer()` 调用过滤链表发送数据, 由模块 `ngx_http_write_filter_module` 负责数据的缓冲和限速, 当所有数据都发送完毕后就结束请求。

结束请求在 HTTP 机制里是一个非常重要的操作, 十分复杂, 因为在请求的每个处理环节中都有可能因为各种原因而停止处理。结束请求的关键是请求结构体里的引用计数 `count`, 它标志了请求上关联操作的数量, 只有当 `count==1` 时才能够真正地结束请求, 其他情况下只能减少引用计数。

HTTP 机制还处理了 `keepalive` 和 `lingering_close` 特性, 如果有这样的要求, 那么就不能直接关闭连接: 前者需要清理资源复用连接, 后者则是延迟一小段时间后再关闭连接。

由于在第 7 章、第 8 章等章节已经介绍了很多 HTTP 机制相关的知识, 所以本章没有再做更多的重复说明, 主要使用了源码加流程图的形式进行讲解, 请读者见谅。

第 18 章

Nginx 与设计模式

通过前面章节的阅读，读者应该对 Nginx 有比较深入的认识，熟悉它的核心运行机制，具备开发 Nginx 各种功能模块的能力。本章将从设计模式的角度对 Nginx 做一个总结。

18.1 设计模式简介

Nginx 是一个 Web 服务器，但它同时又是一个开发框架（Framework），允许开发者将各种功能嵌入这个框架里，所以它必然应用了大量的设计模式以保证系统拥有良好的架构，掌握这些设计模式的用法可以更好地理解 Nginx 的“Why”。

传统意义上的设计模式主要用于面向对象的软件系统，但其解决问题的思想是共通的。虽然 Nginx 的开发语言是 C，但这并不妨碍 Nginx 使用设计模式，而且这些模式相互嵌套，紧密联系，增强了整个 Nginx 框架的扩展性和灵活性。

经典的设计模式有 23 个，分为创建、结构和行为三大类，此外还有很多其他的设计模式，本章按照应用的重要程度列出 Nginx 里使用的设计模式。

18.2 框架级别的模式

这一级别的模式从宏观上搭建了 Nginx 的核心框架，决定了 Nginx 的整体架构体系。

反应器（Reactor）

反应器模式是一种 I/O 事件处理的设计模式，它管理多个事件源，并通过多路分离器把就绪（ready）事件分发到相应的 handler，可以不使用多线程提供高性能的并发处理能力。

Linux、FreeBSD 系统里的 `epoll`、`kqueue` 等系统调用都是反应器模式的具体应用，是高性能 Web 服务器的实现基础。Nginx 也正是利用了 `epoll`、`kqueue` 才能够无阻塞地处理海量的并发连接，所以反应器模式是 Nginx 快速高效的最根本秘密。^①

包装外观 (Wrapper Facade)

包装外观模式整理底层的系统 API，分类、重命名或者简单包装，最后给出一个统一易用的接口，它可以屏蔽不同操作系统之间的差异，增强软件的可移植性。

为了能够跨平台运行，Nginx 大量应用了包装外观模式，使用宏、函数等手段重定义了许多系统函数，减少了 UNIX 平台实现细节差异可能产生的影响。

桥接 (Bridge)

桥接模式分离了架构的设计与实现，架构是稳定的，而实现可以任意变化，增强了系统的灵活性。

Nginx 的模块架构就应用了桥接模式，它使用 `ngx_module_t` 定义模块，结构体里有若干函数指针和扩展字段，然后桥接实现了丰富多彩的 `core`、`conf`、`event`、`stream` 和 `http` 等功能模块，搭建起整个 Nginx 框架。

模板方法 (Template Method)

模板方法模式确定了操作的主要步骤和流程，并在关键节点定义了回调函数，允许外界实现回调函数来扩展或增强原操作的功能，是框架设计中最常用最基本的模式。

作为开发框架，Nginx 在配置解析阶段、请求处理阶段都定义了数量繁多的回调函数，模块可以根据自身的需求实现特定的回调函数，从而插入 Nginx 框架，成为框架的有机组成。

策略 (Strategy)

策略模式封装各种算法，把算法变成一个个接口一致的组件，在运行时能够互相替换，改变行为的内部逻辑。

策略模式在 Nginx 里最明显的应用就是 `load-balance` 模块，它封装了不同的负载均衡算法，而各种 `handler/filter/upstream` 模块其实也可以算是策略模式，在配置文件里替换不同的模块就可以改变 Nginx 的行为，产生不同的响应数据。

^① 与反应器模式对应的是前摄器模式 (Proactor)，它分发的是完成事件 (complete event)。

18.3 业务级别的模式

业务级别应用的模式要比框架级别的模式低一个层次，它们工作在框架内部，决定了某个较具体的功能的设计结构。

对象池（Object Pool）

对象池模式是一个特殊的工厂模式，它预先从系统中生产出一些对象提供给外部，对象在使用完后又放回池中等待下一次使用，可以摊平对象构造和销毁的成本，提高运行效率。

Nginx 里的内存池是对象池模式的典型应用，它从操作系统里申请足够的内存，之后所有的内存分配都在池里进行，最后一次性释放，有效地减少了系统调用次数，也避免了内存泄漏的风险。

`ngx_pool_t/ngx_http_request_body_t` 等结构里也有对象池模式的应用，它们使用一个链表保存闲置的 `ngx_chain_t` 对象，当要使用缓冲区时就直接从链表里取出对象重复利用，无须创建新的对象。

Nginx 进程机制和多线程机制里也应用了对象池模式，就是进程池和线程池。

职责链（Chain of Responsibility）

职责链模式把许多对象连成一个链表，链表里的每一个对象都有机会处理请求，对象彼此独立，既可以协作也可以竞争对请求的处理，非常灵活，是处理数据时常用的设计模式。

Nginx 的 Stream/HTTP 框架里实现了处理引擎和过滤引擎，是职责链模式的具体应用，它们把 `stream/http` 模块组织成链表，逐个地加工处理客户端请求。不过处理引擎更加复杂，它是二维形式的职责链，分为多个阶段，每个阶段里的模块也组织成职责链，而且在处理过程中因为 URI 重写的原因还可以在链表里跳转，导致职责链从头执行。

命令（Command）

命令模式把请求封装为一个对象，让对象携带尽可能多的相关信息，可以简化后续的处理操作，通常配合职责链模式一起使用。

很明显，Nginx 在处理 HTTP 请求时的 `ngx_http_request_t` 结构体就是命令对象，它存储了非常多的信息，如连接、配置、数据、状态、变量、子请求等，在各个模块之间反复传递，由模块职责链进行处理。

备忘录 (Memento)

备忘录模式捕获一个对象的内部状态，并在对象之外保存这个状态作为“备忘”，这样当对象再次启动时就可以通过备忘录恢复原状态，无差错地继续运行。

Nginx 在请求结构体里设计的 `ctx` 成员就是备忘录，它为每个模块提供了保存运行时数据的空间，模块可以把任何数据作为“备忘”存储在 `ctx` 里，不会因为框架的异步机制而导致运行状态不一致。

中介者 (Mediator)

中介者扮演“中间人”的角色，系统里的每个对象只与中介者通信，简化对象之间的多对多联系，协调保证它们共同工作。中介者模式的缺点是它必须维护所有的连接关系，很容易造成自身过度复杂。

`ngx_http_upstream_module` 里定义的 `upstream` 框架就是中介者模式（同时也应用了模板方法模式），它协调 `load-balance` 模块和 `upstream` 模块共同工作，获取上游服务器的地址，然后转发下游的请求和上游的响应数据。它也具有中介者模式的缺点，内部逻辑比较复杂，不太容易理解。

18.4 代码级别的模式

代码级别的模式与具体的代码实现相关，更多地偏向微观的编写程序技巧。

组合 (Composite)

组合模式定义了对象的层次关系，可以把对象组合成树形结构，许多对象不断组合就可以得到一个很大的复合对象，但这个大对象在外部表现和内部行为等方面与小对象完全相同。

Nginx 里的子请求设计就是组合模式，请求结构体使用 `main`、`parent` 和 `posted_requests` 等指针组织成了一个请求树，根是主请求，下面的分支是各个子请求。但从外部来看，一个主请求和子请求是没有任何区别的，可以一致处理。

观察者 (Observer)

观察者模式定义了对象间一对多的联系，当被观察的对象状态变化时观察者对象能够立即得到通知。

Nginx 里的父请求与子请求的通信就应用了观察者模式。子请求设置父请求的处理函数，子请求处理结束时在 `ngx_http_finalize_request()` 中发送通知，这样父请求就能够及

时被唤醒从而继续运行。

适配器 (Adapter)

适配器模式是对接口的包装，它把原接口适配为一个新的接口再插入系统，无须改动任何一方就能够复用原代码。

Nginx 在 event 模块里使用了适配器模式，把 `epoll`、`kqueue`、`select` 等不同的异步 IO 接口统一适配为 `ngx_event_actions_t` 结构体。

原型 (Prototype)

原型模式是工厂模式，可以从一个已有的对象复制出一个相同或者相似的新对象。

Nginx 在创建子请求时使用了原型模式，它从父请求里拷贝了大部分的字段，创建了一个基本相同的新请求对象。

访问者 (Visitor)

访问者模式解耦数据和访问数据的操作，可以在不改变数据的前提下任意增加访问它们的新操作，两者也可以独立变化。

Nginx 变量机制就应用了访问者模式，外界不能直接操作模块的内部数据，只能通过变量提供的 `get/set` 函数来间接访问。所以可以很容易地增加变量来暴露更多的信息，也可以随时变动模块的内部实现，而对外导出的变量则不受影响。

工厂方法 (Factory Method)

工厂方法模式把对象的创建工作封装在一个函数里，改变函数就能改变生产出的对象。

Nginx 创建模块配置数据结构的函数指针 `create_main/srv/loc_conf` 是工厂方法模式，每个模块都可以实现自己的工厂函数，创建出模块专属的配置数据结构。

代理 (Proxy)

代理模式包装一个对象，目的是控制对象的访问，只有通过代理对象才能与被包装的对象通信。

`ngx_str_t` 和 `ngx_buf_t` 代理了一块内存空间，表示一个字符串或者数据块，前者是只读字符串，而后者允许对数据块做更多的操作。

装饰 (Decorator)

装饰模式是另一种形式的包装，它不适配接口，而是在原接口的基础上增加新的功能（当然也可以减少功能），允许动态组合。装饰模式与适配器、代理很像，但目的、用法截然不同。

Nginx 用来连接后端服务的结构体 `ngx_peer_connection_t` 就是装饰模式，它“装饰”复用连接对象 `ngx_connection_t`，添加了后端服务器的地址、连接时间等相关信息。

空对象 (Null Object)

空对象模式是一个可用的对象，但内容和操作都是空的、没有意义的，是空指针概念的强化，最常见的应用是作为“哨兵”表示数组或者链表的结尾。

Nginx 里的指令定义数组、变量定义数组都使用了空对象模式，用一个成员全是 0 的对象作为数组的最后一个元素，这样在遍历数组时就无须知道数组的长度，检测到空对象“哨兵”就是数组的结束。

18.5 总结

本章简略论述了 Nginx 里应用的近二十个设计模式，并分类为框架级别、业务级别和代码级别三类。其实这个分类标准并不是非常精确，有的模式可能跨越了两个级别（例如职责链和中介者），很难说它就明确地属于业务或框架级别。还要补充的是，分类只是依据模式在 Nginx 里应用的层次，并不是说某个设计模式就只能应用在框架、业务或者代码级别，实际上设计模式的应用没有任何限制，层次可高可低，作用域可大可小。

作为一个设计精巧的软件系统，Nginx 里模式应用得非常紧凑，很多模式是相互重叠的。例如 Nginx 在处理 TCP/HTTP 请求时使用了职责链模式，职责链里的每一个节点又是策略模式，并使用备忘录模式暂存数据，在职责链里传递的是命令模式，而这个命令又应用了组合模式。多个模式紧密结合，让 Nginx 架构拥有了良好的扩展性和灵活性，值得我们深入体会其使用方法。

有一些设计模式没有在 Nginx 里出现，如状态、迭代器等，这是因为它们具有较明显的面向对象特征，在 C 语言里实现较困难。

第 19 章

OpenResty 开发

模块化架构决定了 Nginx 的高度扩展性，我们可以使用 C/C++ 语言编写任意的功能模块来增强它的功能，把 Nginx 变成一台全能的应用服务器。但使用 C/C++ 实现 Nginx 模块对开发者的要求较高，我们必须遵循 Nginx 的开发规范，了解工作原理和内部处理流程，熟悉各种数据结构和函数，并在恰当的时机使用它们，程序员必须很好地掌握 Nginx 的进程、事件等运行机制才能开发出一个较为完善的模块。而且由于 C/C++ 的静态编译语言特性，开发周期长、成本高，很难快速迭代。

由 agentzh 创立的开源项目 OpenResty 成功地把 Lua 语言嵌入了 Nginx，用 Lua 作为“胶水语言”粘合 Nginx 的各个模块和底层接口，以脚本的方式直接实现复杂的 HTTP/TCP/UDP 业务逻辑，降低了 Web Server——特别是高性能 Web Server 的开发门槛。^①

OpenResty 基于 Lua5.1/LuaJIT，充分利用了 Lua 内建的协程特性，可以无阻塞地处理并发连接，而且功能代码不需要编译，可以就地修改脚本并运行，简化了开发流程，加快了开发和调试的速度，同时也缩短了开发周期，在如今这个快节奏的时代里弥足珍贵。

很多国内外大型网站都在使用 OpenResty 开发后端应用，而且越来越多，知名的有 Adobe、CloudFlare、Dropbox、GitHub 等，充分地证明了 OpenResty 的优秀。

本章将详细地介绍关于 OpenResty 的方方面面，带领读者一起领略它的风采。

19.1 简介

在官网 (<http://openresty.org>) 上对 OpenResty 的定义是：

^① agentzh 的真名是“章亦春”。

“OpenResty 是一个基于 Nginx 与 Lua 的高性能 Web 平台，其内部集成了大量精良的 Lua 库、第三方模块以及大多数的依赖项。用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。”

从这段描述里我们可以知道，OpenResty 以 Nginx 为核心，集成打包了众多的侧重于高性能 Web 开发的外围组件，它既是一个 Web Server，更是一个成熟完善的开发环境。

做个不太恰当的比方：如果把 Nginx 看作是 Linux，那么 Nginx Plus 就相当于商业版本的 RHEL，质量高但价格昂贵，而 Openresty 则相当于社区版的 CentOS，免费但功能上也毫不逊色，图 19-1 大致表示了这三者的关系。

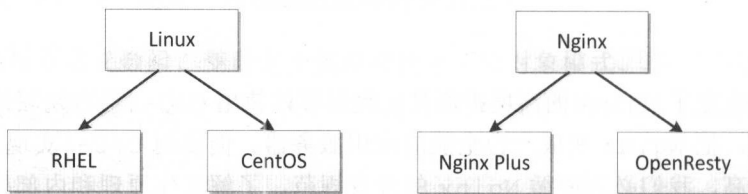


图 19-1 OpenResty 与 Nginx、Nginx Plus 的关系

简单地说：OpenResty 是一个“比 Nginx 更好的 Nginx”。

19.1.1 历史

2007 年，受到当时风行的 OpenAPI 和 REST 潮流的影响，agentzh 使用 Perl 语言^①（还有少量的 Haskell）开发出了一套 Web Service 框架，也就是如今 OpenResty 的雏形。由于 Perl 语言自身的限制，虽然 agentzh 做了大量的优化工作，但性能始终无法令人满意。

2009 年，在综合比较了 Apache、Lighttpd 和 Nginx 等服务器框架的优劣之后，agentzh 决定以 Nginx 作为新的开发平台，与同事 chaoslawful 合力用 C 语言重新设计和实现了之前的 Web Service 框架，并选择小巧紧凑的动态脚本语言 Lua 作为上层的用户语言。就这样，我们所熟悉的高性能服务器开发包 OpenResty 诞生了。^②

2011 年，随着 OpenResty 的用户逐渐增多，开源项目与本职工作的冲突越来越严重，agentzh 于是辞职在家，专心维护 OpenResty，为全世界的程序员提供“免费服务”。

2012 年，旧金山的一家公司向 agentzh 发出邀请，支持他以全职状态继续开发 OpenResty。没有了后顾之忧，agentzh 全心投入了开源事业，为 OpenResty 增加了大量

^① Perl 语言一直是 agentzh 的“最爱”，很多 OpenResty 工具都是用 Perl 编写的。

^② chaoslawful 的真名是“王晓哲”，现在已经逐渐淡出了 OpenResty 的开发。

的新功能，这段时间是 OpenResty 的迅速成长期。

2015 年，第一届 OpenResty 开发大会在北京召开。大会汇集了多个国内外公司和开发者，agentzh 本人也亲自莅临会场，总结回顾 OpenResty 的发展历程，也展望了将来的发展目标。^①

2016 年，OpenResty 软件基金会在香港成立，并获得国内某科技公司 100 万元的捐赠，为 OpenResty 今后持续稳定地前进提供了有力的保证，OpenResty 从此进入了新的发展阶段。^②

19.1.2 版本

OpenResty 的核心组成部分是 Nginx，所以发布版本的命名方式上也“追随”了 Nginx。

OpenResty 使用四位数字作为版本号，形式是：a.b.c.x，其中前三位数字是内部 Nginx 的版本，作为大版本号，第四位数字是 OpenResty 自己的发布版本号，也是小版本号。

例如，OpenResty 1.11.2.3 表示包内部使用的是 Nginx 1.11.2，是本次大版本的第三个发布版本；OpenResty 1.9.7.5 表示包内部使用的是 Nginx 1.9.7，是第五个发布版本。

虽然 OpenResty 以 Nginx 为核心，但并不会紧跟 Nginx Mainline 的更新频率，而是保持了自己的节奏（因为有很多组件要开发和做兼容性测试），版本的发布周期不是很确定。

由于 OpenResty 每次更新都会增加很多新功能，建议读者及时使用最新的版本。本书使用的 OpenResty 是 2017 年中发布的 1.11.2.3 版。

19.1.3 组成

OpenResty 汇集了众多设计精良的组件，搭建了一个完善的高性能服务器开发环境，基于 OpenResty 可以轻易地开发出支持超高并发的 Web 应用和动态网关。^③

① 本书作者十分荣幸地参与了这次会议，还与 agentzh 探讨了一些技术问题。

② 作者编写本书时获悉 agentzh 也在旧金山成立了 OpenResty Inc 公司，探索 OpenResty 商业化的可能，还全职雇用了 Jemplate、Pegex 的作者 Ingy dot Net。

③ 为了叙述方便，本书约定，用 C 语言实现的 OpenResty 组件名字加“ngx_”前缀，用 Lua 语言实现的 OpenResty 组件名字加“lua_”前缀（不过官方的前缀是 lua-resty，以区分标准的 Lua 库，请读者注意）。

OpenResty 的核心组件有三个，分别是：

- Nginx : 高性能的 Web 服务器（如果读过前几章相信已经很熟悉了）；
- LuaJIT : 高效的 Lua 语言解释器/编译器（Just-In-Time Compiler）；
- ngx_lua : 以模块的方式让 Lua 语言嵌入在 Nginx 里执行。

使用这三个核心组件，OpenResty 就可以完成相当多的网络应用开发工作了，但 OpenResty 远不止如此，它还包含了其他一些非常有用的 Nginx 模块和 lua-resty 库，进一步增加了开发工作的便利，较常用的有：

- ngx_echo : 提供一系列“echo”风格的 Nginx 指令和变量；
- ngx_set_misc : 增强的“set_xxx”指令，用来操作变量；
- ngx_headers_more : 更方便地处理 HTTP 请求头和响应头的指令；
- lua_cjson : C 实现的快速解析 JSON 格式数据的 Lua 库；
- lua_redis : 非阻塞的 redis 客户端；
- lua_memcached : 非阻塞的 Memcached 客户端；
- lua_mysql : 非阻塞的 MySQL 客户端；
- lua_string : md5/sha1/sha256 等字符串功能；
- lua_lrucache : 高效的 LRU 缓存；
- lua_lock : 基于共享内存的非阻塞锁；
- lua_websocket : 使用 Lua 实现的非阻塞 WebSocket 功能；
- lua_healthcheck : 使用 Lua 实现的 upstream 健康检查；
- lua_limit_traffic : 使用 Lua 定制更灵活的流量控制策略。

此外，OpenResty 里还有几个辅助开发、调试和运维的工具，例如：

- opm : 类似 rpm、npm 的包管理工具；
- resty-cli : 以命令行的形式直接执行 OpenResty/Lua 程序；
- restydoc : 类似 man 的参考手册，非常详细。

可见，OpenResty 是一个功能非常完备的服务器开发包，大多数 Web 应用所需的功能都已经包含在了里面，所谓的“out of box”，开发者只需要简单地自己的程序里引用，就能够轻松享用这些高质量的模块和库，从而快速实现新的业务。

OpenResty 的组成可以用图 19-2 来表示。

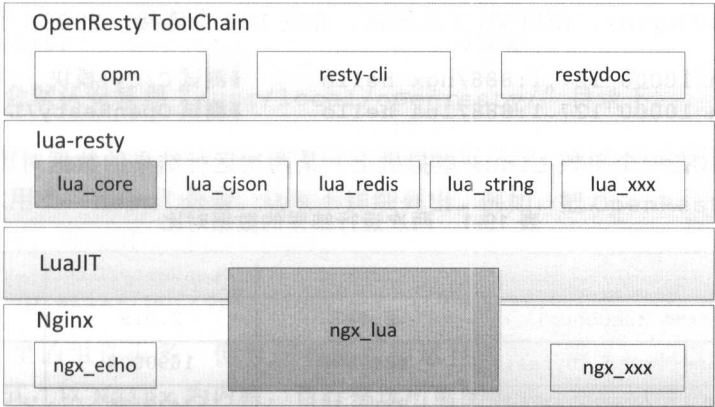


图 19-2 OpenResty 的组成

19.1.4 性能

虽然 OpenResty 基于高性能的 Nginx，也有诸多的成功应用案例，但仍然存在很多人对它抱有疑虑、持观望态度。原因之一是 OpenResty 使用的语言——不是系统编程语言 C/C++，而是动态脚本语言 Lua，“想当然”地认为性能比不上原生的 C 模块，甚至不如 Java。

这实在是个非常严重的误解。

Lua 语言虽然是个脚本语言，但它非常小巧高效，而 OpenResty 内部使用的又是经过高度优化的 LuaJIT 运行环境，使用 JIT 技术编译 Lua 代码，运行效率比原本的 Lua 解释器又高出数倍，完全能够达到媲美 C/C++代码的程度。

我们可以用一个实际的例子来对比验证一下 C/C++模块与 OpenResty/Lua 的运行效率。

在 Nginx 里定义如下的两个 location，分别用 C/C++模块和 Lua 输出一个字符串：

```
server {
    listen      888;
    server_name localhost;

    location /ngx_hello {
        ndg_echo "hello nginx\n";
    }

    location /lua_hello {
        content_by_lua_block {
            ngx.print("hello nginx\n")
        }
    }
}
```

#定义一个对比测试用的 server
#端口号是 888
#服务本机地址

#C/C++模块
#输出一个字符串

#OpenResty/Lua
-- Lua 语言产生响应内容
-- 输出同样的字符串

然后我们启动 Nginx，使用 ab 工具测试，并发 100 个连接，10000 个请求：

```
ab -c 100 -n 10000 127.1:888/nginx_hello      #测试 C/C++模块
ab -c 100 -n 10000 127.1:888/luu_hello      #测试 OpenResty/Lua
```

表 19-1 是（在一个单核 Linux 虚拟机上）某两次运行结果的数据对比：^①

表 19-1 两次运行结果的数据对比

	C/C++	OpenResty/Lua	Ratio
Time taken (seconds)	2.847	2.919	102.53%
Total transferred (bytes)	1290000	1690000	131.01%
Requests per second	3512.27	3425.75	97.54%
Time per request (ms)	28.472	29.191	102.53%
Transfer rate (k/s)	442.46	565.38	127.78%

由表 19-1 中的数据可见 OpenResty/Lua 的运行效率与我们自己编写的 C/C++模块基本相当，在关键指标上的差距不到 3%，有的指标还有所超越——要注意这使用的可是一向被人所轻视的“脚本语言”。

读者也可以再做其他测试，相信更多实测的数据会让你打消顾虑，抛弃成见。

19.1.5 安装

目前 OpenResty 提供正式的 yum 源，RedHat 系的 RHEL、CentOS、Fedora 可以在配置好源后直接使用 yum 安装（具体方法可参见官网），例如：

```
sudo yum install openresty      #直接使用官方 Linux 安装包
```

不过本书还是推荐以源码的方式安装 OpenResty，可以更灵活地定制所需的功能。

安装 OpenResty 的过程与 Nginx 基本相同，下载压缩包，解压后执行 configure 再 make 即可，例如：

```
wget https://openresty.org/download/openresty-1.11.2.3.tar.gz
tar xvfz openresty-1.11.2.3.tar.gz      #解压缩
cd openresty-1.11.2.3                  #进入源码目录

./configure                             #编译前的配置工作
```

^① 测试结果里两者发送的数据不一致的原因是 OpenResty 的响应头多一些。实际上，如果 OpenResty 使用指令“lua_http10_buffering off”，对这种短连接的测试场景性能还可以提高。

```
make #编译
sudo make install #安装
```

OpenResty 会默认安装到 “/usr/local/openresty/” 目录下。

与 Nginx 一样，OpenResty 也可以使用丰富的配置选项，如定制编译参数、安装目录、增减模块等，可以用 “--help” 查看，这里不详细列出。例如，把 OpenResty 安装到 /opt/openresty 目录下，可以用：

```
./configure --prefix=/opt/openresty #编译前的配置工作，指定安装目录
```

OpenResty 里的组件很多，但有的可能实际运行时并不需要，所以我们可以仿造 OpenResty 的方式，以 Nginx 为内核，自行挑选所需的模块，打造出自己的 “Openresty 发行版”。

自行编译 OpenResty 必须的组件是 Nginx、LuaJIT 和 ngx_lua，篇幅所限不再说明，具体步骤可以参考 GitHub 上的文档。

19.1.6 目录结构

安装之后 OpenResty 的目录结构如下：

/usr/local/openresty/	#OpenResty 安装主目录
├── bin	#OpenResty 的可执行文件
├── luajit	#LuaJIT 的运行库和头文件等
├── lualib	#自带的 lua-resty 库
├── nginx	#完整的 Nginx 环境
├── pod	#参考手册 (restydoc) 使用的数据
└── site	#包管理工具 (opm) 使用的数据

在 bin 目录里存放有几个重要的 OpenResty 可执行文件，它们是：

- opm : OpenResty 的包管理工具；
- resty : OpenResty 的命令行工具，可直接执行 Lua 程序（见 19.1.7 节）；
- restydoc : OpenResty 的参考手册（见 19.1.8 节）；
- openresty : OpenResty 的可执行文件。

最后一个 bin/openresty 是对安装目录里 nginx/sbin/nginx 的符号链接，实际上就是 Nginx，这种做法更好地凸显了 OpenResty，而且屏蔽了内部的目录结构细节，也避免了与系统里可能存在的其他 Nginx 的冲突。

因为 OpenResty 的目录结构比较复杂，不建议修改安装目录里的 Nginx 配置文件和 Lua 代码，也不要直接从安装目录里启动服务，而是要利用 Nginx 的 “-p” 参数，在一个 “干净”

的新环境里启动 OpenResty，在这个新环境里可以任意添加配置文件、组织 Lua 代码结构，不会“污染”OpenResty 安装目录，并且可以很容易地运行多个 OpenResty 实例。

通常我们把以这种方式部署和运行的 OpenResty 实例称为“OpenResty 应用服务”，有时也简称为“OpenResty 应用”或者“OpenResty 服务”。

本书使用的运行环境是“~/openresty”，目录结构如下：^①

~/openresty	#OpenResty 的运行目录
— conf	#OpenResty/Nginx 的配置文件
— logs	#OpenResty/Nginx 的日志目录
— lua	#运行在命令行里的 Lua 代码
— service	#运行在服务器里的 Lua 代码，即服务
— conf	#服务程序的配置
— lib	#服务程序使用的动态库
— resty	#服务程序使用的 lua-resty 库
— http	#HTTP 服务程序
— stream	#TCP/UDP 服务程序
— utils	#通用的实用工具程序

启动 OpenResty 应用的命令是：

```
/usr/local/openresty/bin/openresty -p `pwd` #在本目录启动 OpenResty 服务
```

读者可以依据实际情况调整目录结构，创建符合自己需要的 OpenResty 运行环境。

19.1.7 命令行工具

OpenResty 在 bin 目录下提供一个命令行工具 resty，可以把它作为 Lua 语言的解释器（但运行在 OpenResty 环境里），写出类似 Perl、Python 那样易用的脚本。

resty 的工作原理是启动了一个“无服务”的 Nginx 实例，禁用了 daemon 等大多数指令，也没有配置监听端口，只是在 worker 进程里用定时器让 Lua 代码在 Nginx 里执行。

使用“-e”参数可以在命令行里直接执行 Lua 代码，例如：

```
./resty -e "print('hello lua')" #执行 Lua 代码，打印一个字符串
```

这种方式只适合执行很小的 Lua 代码片段，更好的方式是利用 UNIX 的“Shebang”(!)，在脚本文件里的第一行指定 resty 作为解释器，能够书写任意长度和复杂度的代码，而且更利于管理维护。

^① 本书使用的目录结构分离了 Nginx 配置文件与服务程序（Lua 代码），较易于维护管理。

刚才的命令行用法可以改写成下面的脚本文件：^①

```
#!/usr/local/openresty/bin/resty      -- 使用 resty 作为脚本的解释器
print("hello lua")                    -- 执行 Lua 代码，打印一个字符串
```

脚本文件也支持传递命令行参数，参数存储在表 `arg` 里，可以用 `arg[N]` 的方式访问：

```
#!/usr/local/openresty/bin/resty      -- 使用 resty 作为脚本的解释器

local n = #arg                        -- 得到参数的数量
print("args count = ", n)             -- 打印参数的数量
for i = 1, n do                       -- 变量参数表，注意 Lua 下标从 1 开始
    print("arg ", i, ": ", arg[i])    -- 输出参数
end                                    -- 循环结束
```

使用参数执行脚本 `hello.lua`，结果是：

```
./hello.lua FireEmblem Heroes        #执行 Lua 代码，带两个参数
args count = 2                       #打印参数的数量
arg 1: FireEmblem                    #输出第一个参数
arg 2: Heroes                        #输出第二个参数
```

`resty` 工具还有其他的选项，可以指定 `Nginx` 配置文件、库搜索路径、共享内存、并发数等，读者可以参考 `help` 或者 `GitHub` 上的帮助文档。

本书之后在讲解 `Lua` 语言和 `LuaJIT` 时均采用 `resty` 作为解释器执行 `Lua` 程序。

19.1.8 参考手册

`OpenResty` 附带了非常完善的用户参考手册 `restydoc`，提供与 `UNIX` 手册 `man` 相同的功能，可以检索 `OpenResty` 里所有组件的帮助文档，包括但不限于：

- `Nginx` 介绍、用法、基本工作原理；
- `Nginx` 各个模块的介绍、指令、变量的用法；
- `OpenResty` 各个组件的介绍和用法；
- `OpenResty` 指令和 `API` 的用法；
- `Lua/LuaJIT` 语法要素。

下面示范了一些 `restydoc` 的用法，其中的 “`-s`” 参数用来指定搜索手册里的小节名：

```
restydoc nginx                        #Nginx 总体介绍
restydoc beginners_guide              #Nginx 初学者教程
```

^① 某些系统上可能要使用 “`#!/usr/bin/env /usr/local/openresty/bin/resty`” 的形式。

restydoc events	#Nginx 事件机制的说明
restydoc ngx_http_access_module	#Nginx 访问控制模块的说明
restydoc ngx_lua	#ngx_lua 模块的说明
restydoc ngx_echo	#ngx_echo 模块的说明
restydoc luajit	#LuaJIT 的说明
restydoc opm	#opm 的说明
restydoc -s args	#变量\$args 的说明
restydoc -s proxy_pass	#proxy_pass 指令的说明
restydoc -s content_by_lua_block	#content_by_lua_block 指令的说明
restydoc -s ngx.say	#OpenResty 接口 ngx.say 的说明
restydoc -s concat	#Lua 函数 concat 的说明

多使用 restydoc 可以帮助我们尽快熟悉 OpenResty 开发。

19.2 Lua 语言

Lua 语言是一种动态脚本语言，与 Perl、Python、Ruby 等语言不同，它设计的目标是要能够嵌入其他应用程序里，提供脚本化的扩展和定制功能，所以 Lua 是一个小巧紧凑而非大而全的语言。

Lua 语言本身只有一个精简的核心和最基本的库，所以标准解释器非常小，只有一百多 K。Lua 代码的执行效率非常高，速度只比 C 语言的实现低大约 10% 到 20%。Lua 天生与 C/C++ 有非常好的互操作性，可以很容易地嵌入 C/C++ 程序并调用宿主函数，从而轻松地扩展功能。

Lua 语言的功能也很丰富，表 (table) 结构十分灵活，可以模拟出其他语言里的数组、集合、字典、类、名字空间等特性，还提供闭包 (即 C++ 里的 lambda 表达式) 支持函数式编程，提供协程 (coroutine) 支持并发编程。

Lua 语言有 5.1、5.2 和 5.3 三个主要的版本，版本之间有一些语法上的差异，不完全兼容，目前 OpenResty 使用的是 Lua 5.1+LuaJIT 扩展。

本节简要介绍 Lua 5.1 的语法和标准库，下一节研究 LuaJIT 的扩展功能。^①

19.2.1 注释

Lua 的注释语法“独树一帜”，与 C 家族的 C++、Java 和 Shell 家族的 Perl、Python 都不同，使用的不是常见的“//”或者“#”，而是连续的两个“-”，也就是“--”。

① 篇幅所限本书只能扼要介绍 Lua 语言里最基本的部分，不可能面面俱到，但有一个很全面的中文网站可供读者参考，网址是：<http://book.luaer.cn/>。

单行注释使用简单的“--”，例如：

```
-- this is a comment
-- 当然也可以使用中文注释
```

```
print("hello lua")                -- 行尾注释
```

多行注释使用“--[[...]]”的形式：

```
--[[
多行注释，非常方便
可以很容易地注释掉大段的代码，或者书写说明文字
]]                                -- 多行注释结束
```

19.2.2 数据类型

Lua 语言提供六种基本的数据类型：

- nil : 类似 C++ 的 nullptr 或 Python 的 None，表示不存在的空对象；
- boolean : 布尔类型，取值为 true 或 false；
- number : 数字类型，Lua 不区分整数和浮点数；^①
- string : 字符串类型，准确地说应该是“字节序列”，允许包含二进制数据；
- function : 函数类型，相当于 C++ 里的函数对象或者“可调用物”；
- table : 表类型，非常灵活的数据结构，可以模拟出数组、字典等其他结构。

使用函数 type() 可以测试变量的类型，它以字符串的形式返回类型的名字，例如：

```
print(type(nil))                -- nil
print(type(true))               -- boolean
print(type(42))                 -- number
print(type(2.718))              -- number
print(type("metroid"))          -- string
print(type(print))               -- function (print 是 Lua 标准库里的一个函数)
print(type(table))              -- table (table 是 Lua 标准库里的一个表)
```

虽然变量是有类型的，但因为 Lua 是动态语言，所以声明变量并不需要显式地写出类型，变量也可以存储任意类型的值：

```
x = 2017                        -- 变量的类型是 number
x = "lua"                       -- 变量的类型变为 string
x = nil                         -- 变量的类型变为 nil
```

^① 这是 Lua 5.1 和 5.2 的语法，Lua 5.3 引入了整数类型，并支持了位运算。

Lua 的字符串形式很灵活，单引号或者双引号都可以，字符串里也可以使用转义符：

```
print('openresty')           -- 单引号形式的字符串
print("It's OK")            -- 双引号形式的字符串，里面可以包含单引号
print("lua\tnginx")          -- 使用转义字符\t
```

Lua 还用 “[[...]]” 的形式支持 raw string，括号内的字符不会转义，在写正则表达式的时候非常方便：

```
print([[raw string \r\n]])    -- 字符串里的\r\n等不会被转义
print([[^\d+.\d+$]])          -- 直接是字符串的“原始形态”
```

还需要补充一点：Lua 里的字符串都是不可修改的（只读），不能像 C/C++ 那样单独访问或改动字符串里的某个字符。

19.2.3 变量

Lua 语言里的变量有作用域（scope）的概念，分为局部变量和全局变量。

局部变量需要使用关键字 “local” 声明，作用域仅限本代码块（文件内或语句块内），没有关键字 “local” 声明的变量都是全局变量，而且不需要声明就可以直接使用：^①

```
x = 1                        -- 使用一个全局变量 x，赋值为 1，全局可见
local str = 'matrix'         -- 使用一个局部变量 str，仅此文件内可见

do                            -- 开始一个代码块
    local pai = 3.14          -- 局部变量 pai，仅在此代码块内可见
end                            -- 代码块结束

print(type(pai))             -- 局部变量 pai 消失，访问的是全局变量 pai
```

一个变量如果没有显式赋值，那么它的值就是 nil，所以代码的最后一行会输出 “nil”。

在 Lua 里应当尽量少使用全局变量，多使用局部变量。局部变量不仅很好地控制了变量的作用域，避免全局名字冲突，而且因为“局部化”，解释器查找的速度也更快。

^① 可以把 local 近似地理解为 C++ 里的 static+any。

19.2.4 运算

Lua 支持基本的加减乘除和取模运算，幂运算使用符号“^”：^①

```
print(1 + 1, ",", 5 - 3)      -- 加减法运算，输出 2,2
print(2 * 4, ",", 1 / 3)      -- 乘除法运算，输出 8,0.33333333333333
print(5 % 2, ",", 3 ^ 3)      -- 取模和幂运算，输出 1,27
```

Lua 不提供 C/C++ 里的递增和递减操作符（可能的原因是“--”已经被用作注释语法了），递增递减操作只能使用标准的赋值形式，例如：

```
count = 10                    -- 一个整数变量
count = count + 1             -- 加 1 后赋值，相当于 ++count
```

Lua 的关系运算符与 C/C++ 基本相同，但不等比较使用的是“~=”，需要特别注意：

```
print(3.14 > 2.718)           -- 大于关系
print(1/3 == 2/6)             -- 等于关系
print('10' ~= 10)            -- 不等关系，注意运算符是“~=”，不是“!=”
```

在执行比较操作时 Lua 首先比较变量的类型，如果类型不同则直接返回 false。实际开发中比较常见的情况是比较字符串形式的数字，必须用函数 `tonumber()`/`tostring()` 显式转换数字类型或字符串，例如：

```
print(tonumber('10') == 10)   -- 等于关系
```

Lua 的逻辑运算符有 `and`、`or` 和 `not` 三个，但运算规则有些特殊：^②

- `nil` 和 `false` 认为是假，其他都是真，包括数字 0；
- `x and y`，如果 `x` 是真，返回 `y`，否则返回 `x`；
- `x or y`，与 `and` 操作正好相反，如果 `x` 是真，返回 `x`，否则返回 `y`；
- `not x`，只返回 `true/false`，对 `x` 取反。

对于 C/C++ 程序员来说第一条规则需要特别注意，因为在 C/C++ 语言里 0 是假，和 `false` 等价，但在 Lua 语言里 0 是真，`nil` 和 `false` 等价，在书写条件判断语句时必须小心。

下面的代码示范了 Lua 里的逻辑运算：

```
print(0 and 'abc')            -- 0 是真，所以返回字符串 'abc'
print(x or 100)               -- x 是全局变量，值是 nil，返回 100
```

① 数学运算可以混用字符串类型，Lua 会自动把字符串转换为数字，但如果含有非数字字符无法转换就会出错，所以最好不要混用。

② 可能很多人都不知道，C++ 的逻辑运算也支持使用 `and/or/not`，代替 `&&/||/!`。

```
print(not x)                -- x 是全局变量，值是 nil，返回 true
```

利用 Lua 逻辑运算的特性可以实现非常灵活的赋值功能：

```
local x = count or 100      -- 为 x 赋初值，count 不存在则默认值是 100
local y = a and b or c      -- 相当于 a?b:c，由 a 的真假决定赋值 b 或 c
```

Lua 内建支持字符串连接操作，使用运算符 “.”：

```
print('hello'..' '..'world') -- 连接多个字符串
```

虽然 Lua 可以高效地处理字符串，但字符串连接操作应当尽量少用，因为每一次字符串连接就会创建一个新的字符串对象，如果多次操作超长字符串（例如几 M 甚至几十 M 的大块数据）很容易就会导致 LuaVM 内存耗尽，发生错误。

计算字符串的长度可以用特别的运算符 “#”：

```
print(#'openresty')        -- 计算字符串长度，输出 9
```

最后我们需要注意操作数是 nil 的情形，很多时候对 nil 运算都会导致错误，例如：

```
x = nil                    -- x 的值是 nil
print(1 + x)               -- 出错，无法执行加法运算
print("msg is "..x)        -- 出错，无法执行连接运算
```

如果一个变量可能是 nil，最好使用 or 运算给它一个默认值：

```
print(1 + (x or 2))        -- 正常，x 不存在则使用 2 执行加法运算
print("msg is ".. (x or "-")) -- 正常，x 不存在则使用 "-" 执行连接运算
```

19.2.5 语句

Lua 里的语句包括赋值语句、条件判断语句和循环语句。

Lua 语句的格式非常自由，不强制要求缩进。语句末尾可以使用 “;” 表示结束，但不是必需的，实际编写代码时通常都省略。

赋值语句

赋值语句是编程语言最基本的语句，Lua 使用 “=” 在变量里存储一个确定类型的值。

除了最基本的形式，Lua 还允许用逗号分隔，在一个语句里声明或赋值多个变量，这是个非常便利的特性：

```
local data, err            -- 声明了两个局部变量，默认值都是 nil
local a, b = 1, 'lua'      -- a、b 分别赋值为数字 1 和字符串 'lua'
local x, y = a, b, 'not_used' -- x、y 分别赋值为 a 和 b，第三个字符串未使用
```

```
local m, n = "only this"      -- m 被赋值为字符串, n 没有被赋值, 所以是 nil
```

语句块

使用“do ... end”的形式可以声明一个语句（代码）块，里面包含任意多条语句，相当于 C/C++ 里的花括号复合语句，例如：

```
do                                -- 开始一个语句块
    local x = 10                 -- 一个局部变量, 仅本块内有效
    print("x = ", x)            -- 输出变量的值
end                              -- 语句块结束
```

条件判断语句

Lua 没有 C/C++ 里的 switch-case 语句，只有 if-else，形式是：

```
if conditions then               -- 条件判断
    ...                          -- 执行语句
elseif conditions then          -- 其他条件判断
    ...                          -- 执行语句
else                            -- 判断条件都不满足进入这里
    ...                          -- 执行语句
end                              -- 条件判断语句结束
```

需要注意的是 else-if 语句的写法，不是 C/C++ 里的“else if”或者 Python 里的“elif”，而是“elseif”（中间没有空格）。^①

while/until 循环语句

Lua 的 while/until 循环与 C/C++ 的 while/do-while 类似：

```
while conditions do             -- while 循环语句开始, 条件成立时执行循环体
    ...                         -- 执行语句
end                             -- while 循环语句结束

repeat                          -- until 循环语句开始
    ...                         -- 执行语句
until conditions                -- until 循环语句结束, 条件成立时退出循环体
```

注意 until 语句的条件判断，与 C/C++ 的 do-while 意义是相反的，当条件成立时退出循环。

^① 作者不建议过多地使用 else/elseif，它增加了代码的逻辑分支和缩进层次，导致难以维护的代码。

for 循环语句

Lua 的 for 循环语句有两种形式：数值循环和范围循环，这里先介绍前者，后者在 19.2.7 节讲解。

for 的数值循环类似 C/C++ 的标准 for 语句，但形式上要简洁一些：

```
for var=m,n,step do      -- for 循环语句开始
...                      -- 执行语句
end                      -- for 循环语句结束
```

语句的含义是：变量 var 从 m 前进（或后退）到 n，执行循环体里的语句，参数 step 是用于控制 var 前进或后退的步长，可以省略，默认是 1。例如：

```
for i=1,5 do             -- for 循环语句开始，从 1 到 5
    print("for : ", i)   -- 输出数字
end                     -- for 循环语句结束
assert(not i)            -- for 里的 i 是局部变量，循环外自动失效
```

for 循环里的变量 var 会自动声明为局部变量（虽然没有使用 local 关键字），而且仅在 for 语句里有效。

循环控制语句

在循环语句里可以用 break 或者 return 语句直接退出循环，用法与 C/C++ 相同，但必须在语句块的最后——也就是后面紧跟着 end/until 关键字。如果想要在任意的位置结束循环，可以使用“do break end”的形式。

Lua 语言不提供 continue 语句，但 OpenResty 使用的 LuaJIT 扩展支持 goto，可以变通实现 continue，将在 19.3.1 节介绍。

19.2.6 函数

在 Lua 语言里函数是一类特殊的变量，它持有一个语句块，使用参数执行语句块，然后返回结果。

定义函数需要使用关键字 function，形式是：

```
[local] function func_name(arguments)  -- 定义一个函数
...                                    -- 函数体
end                                    -- 函数定义结束
```

这种形式实际上是函数变量声明的简化形式，相当于：


```
[local] func_name = function(arguments)    -- 定义一个函数
    ...                                    -- 函数体
end                                          -- 函数定义结束
```

可以看到，Lua 的函数就是变量，也可以使用 local 局部化。也正是因为函数是变量，所以 Lua 里不存在 C/C++ “前向声明”的概念，任何函数都必须定义后才能使用。

Lua 函数的参数和返回值都非常灵活。入口参数并不受声明的限制，可以传入任意数量的实参，少的默认值是 nil，多的则被忽略；函数的返回值使用 return 语句，可以用逗号分隔返回多个值。

Lua 函数的参数都是传值（但表除外），也就是说函数体内的修改不会改变传入参数的值。

下面的代码示范了 Lua 函数的用法：

```
local function f1(a)                                -- 定义函数，参数数量是 1
    a = 'sliver'                                     -- 修改参数的值，但不会影响外部变量
    print("var is ", a)                             -- 输出变量的值
end                                                  -- 函数定义结束

local x = 'golden'                                  -- 声明一个局部变量
f1(x, 'heart')                                       -- 调用函数，多传入的参数被忽略
assert(x == 'golden')                               -- 外部变量没有被修改

local function f2(a, b, c)                           -- 定义函数，参数数量是 3
    print(a .. (b or '') .. (c or ''))              -- 输出变量，使用 or 运算防止 nil
end                                                  -- 函数定义结束

f2('Crazy', 'Diamond')                             -- 调用函数，少的参数默认值为 nil

local function f3(a, b)                               -- 定义函数，参数数量是 2
    return a+b, a*b                                  -- return 返回多个结果
end                                                  -- 函数定义结束

local x, y = f3(10, 20)                             -- 使用赋值语句接收返回的多个结果
```

Lua 函数本身也是“闭包”(Closure)，函数一旦声明就捕获了之前的所有变量——这些变量在 Lua 里被称为 upvalue，然后可以把函数存储在变量里任意传递和使用，如果读者熟悉 C++ 里的函数对象或者 lambda 表达式相信对这种用法不会陌生。

函数的闭包用法属于 Lua 语言里较高级的技巧，读者可在实际工作中逐渐学习体会，这里就不再举例说明了。

19.2.7 表

表 (table) 是 Lua 里唯一的数据结构, 可以近似地理解与其他编程语言里的字典、关联数组或者 key-value 映射, 但 Lua 的表更加灵活, 能够构造出任意复杂的数据结构。

Lua 表里作为索引的 key 可以是任何非 nil 值, 所以当 key 类型是整数时表就相当于数组, key 类型是字符串时表就相当于字典或关联数组。Lua 表对 value 的类型没有任何限制, 当然也可以是另外一个表, 从而实现多个表的嵌套。

Lua 里定义表使用花括号 “{}”, 访问表元素使用方括号 “[]”, 如果 key 是字符串也可以直接使用点号 “.” 来访问, 这时表就可以模拟其他语言里的类/对象或名字空间特性, 存储成员变量和成员函数:

```
local a = {3, 5, 7}           -- 声明一个数组形式的表
assert(a[1] == 3 and a[2] == 5) -- 使用 [] 访问数组里的元素, 整数 key

local x = {}                 -- 声明一个空表
x['name'] = 'samus'          -- 使用 “[]” 访问表里的元素, 字符串 key
x.job = 'hunter'             -- 使用 “.” 访问表里的元素, 字符串 key

print(x.name, ' : ', x['job']) -- “.” 和 “[]” 的方式可以随意切换

x.mission = function(dst)    -- 为表添加一个函数变量, 相当于成员函数
    print('fly to ', dst)    -- 输出一个字符串
end                          -- 函数定义结束

x.mission('zebes')           -- 执行成员函数
```

要特别留意的是: 当表作为数组来使用时整数下标索引必须从 1 开始计数, 这是与 C/C++/Python 等语言最大的不同。^①

计算表里的元素数量可以用运算符 “#”:

```
assert(#a == 3)              -- 计算表里的元素数量, 输出 3
for i=1, #a do               -- 用 # 计算数组长度, 遍历数组
    print(a[i], ', ')       -- 输出数组元素
end                          -- for 循环结束
```

遍历表里的元素还可以使用 for 循环语句的第二种形式, 即范围循环, 但需要配合两个

① 实际上 Lua 数组的索引可以从任意的整数值开始, 但从 1 计数已经成为了 Lua 世界里“不成文”的约定, 如果强制使用其他的计数方式将导致有的标准库无法正常工作。

标准库函数: `ipairs()` 和 `pairs()`。这两个函数起到迭代器的作用, 返回表里的 `key/value`, 前者只适合数组形式的表, 并且遍历到 `nil` 值就结束, 而后者可以支持任意形式的表, 并且遍历表里的所有元素, 但速度没有 `ipairs` 快。

使用 `for+ipairs/pairs` 遍历表的示范代码如下:

```
for i,v in ipairs(a) do          -- ipairs() 只能遍历数组形式的表
    print(a[i], ' ', v)         -- 输出数组元素
end                               -- for 循环结束

for k,v in pairs(x) do           -- pairs() 可以遍历任意的表
    print(k, ' => ', v)         -- 输出数组元素
end                               -- for 循环结束
```

因为表通常都很大, 所以当表作为函数的参数时不是传值的方式, 而是传引用的方式, 也就是说没有拷贝, 直接传递表的“引用”, 函数体内可以直接修改表的元素, 例如:

```
local function f(v)              -- 定义函数, 表传递的是引用
    v.name = v.name .. ' aran'   -- 函数内部可以直接修改表的元素
end                               -- 函数定义结束

f(x)                             -- 执行函数, 表被修改
print(x.name)                    -- 输出'samus aran'
```

19.2.8 标准库

Lua 的库实际上就是包含了函数成员的表, 这里表起到了名字空间的作用。

Lua 标准库很小, 只提供基本的功能:

- `base` : 最基本的函数, 如 `type`、`print`、`pairs`;
- `string` : 字符串相关函数, 如取子串、格式化、大小写转换等;
- `table` : 表相关函数, 如插入删除元素、排序等;
- `math` : 数学计算相关函数, 如三角函数、平方根等;
- `io` : 文件相关函数, 如打开、关闭、读写文件, 注意是阻塞的;
- `os` : 操作系统相关函数, 如时间日期、创建目录等;
- `debug` : 调试用的函数。

标准库函数的详细接口说明都可以在 Lua 或 OpenResty 手册里查阅, 本节只简略介绍一些较常用的。

`string.sub(s, from, to)` 可以提取子串, 字符串的计数从 1 开始, 也可以使用负数,

-1 表示字符串的末尾，例如：

```
local str = 'hello lua'           -- 一个字符串
print(string.sub(str, 1, 4))      -- 取子串'hell'
print(string.sub(str, 7))        -- 取子串'lua'
print(string.sub(str, 5, -1))     -- 取子串'o lua'
```

`string.byte(s, from, to)` 用来提取字符串里的字节，返回若干个整数：

```
local a,b = string.byte(str, 1, 2) -- 取字符串里的前两个字节，即'he'
print(a, " ", b)                  -- 输出 104 和 101，即'he'的 ASCII 码值
```

`string` 库里还提供模式匹配/替换功能的 `find/gfind/gsub` 等函数，但使用的不是标准的正则表达式语法，而且速度也不快，不建议在代码里使用，而是应该用 OpenResty 自己的 `ngx.re` 系列函数，它基于 PCRE 库，性能很高。

`table.concat(arr, sep)` 可以把数组 `arr` 里的元素用分隔符 `sep` 连接为一个字符串，比用 “..” 运算符速度要快，也更节约内存，例如：

```
local a = {'openresty', 'lua'}    -- 一个字符串数组
print(table.concat(a, '+'))        -- 使用 '+' 连接，结果是 'openresty+lua'
```

`table` 库里还有 `insert` 和 `remove` 函数可以添加删除表里的元素，但效率并不高，对于数组形式的表，可以用如下的方式高效添加元素：

```
a[#a + 1] = 'nginx'               -- 利用 “#” 运算符获取长度来添加元素
```

`debug` 库提供各种调试用的函数，最好不要用在正式的生产代码里，其中比较有用的一个函数是 `debug.traceback()`，它输出函数的调用栈，可以打印在日志里追踪 Lua 代码的执行情况：

```
print(debug.traceback())          -- 输出 Lua 代码在此处的调用栈信息
```

实际开发中一个使用库函数的技巧是把它们 `local` 化，为函数起“别名”，可以避免 Lua 解释器反复在表里查找函数，加快程序的执行速度，例如：

```
local str_sub = string.sub        -- 用 str_sub 代替 string.sub
local concat = table.concat       -- 用 concat 代替 table.concat

print(str_sub(str, 1, 4))          -- 取子串
print(concat(a, '+'))              -- 使用 '+' 连接
```

19.2.9 模块

Lua 语言基于表实现了与 C/C++ 的 `namespace`、Java 的 `package` 类似的模块机制，

用户可以用模块来管理组织代码结构，Lua 标准库和 OpenResty 里的所有功能也都使用了这种方式。

模块就是一个函数库，表现为一个 Lua 表，里面有模块作者提供的各种功能函数，可以用点号 “.” 访问。

使用 `require` 函数可以加载模块，参数是模块所在的文件名，需要用一個变量来保存 `require` 函数的返回结果——通常是一个表。例如：^①

```
local cJSON = require "cjson"      -- 加载 OpenResty 的 cJSON 模块，local 化

local str = cJSON.encode({a=1,b=2}) -- 调用模块里的函数，JSON 编码
print(str)                          -- 输出: {"a":1,"b":2}
```

编写自己的模块也很容易，在代码文件里创建一个表，把函数作为表的元素，最后用 `return` 返回这个表就可以了。示范代码如下：^②

```
local proto = {                    -- 定义一个表，包含模块的所有功能
    version = '0.1'                -- 给一个基本的版本号信息
}

function proto.run()               -- 定义一个函数，注意不能是 local
    print("run in mod")            -- 函数体
end                                -- 函数定义结束

return proto                       -- 返回模块的表
```

当然，要开发出真正可用的模块远不止这么简单，还需要其他一些知识（例如 `self` 参数、“.” 操作符等），篇幅所限本书不展开详述，读者可在书后查找相关资料学习。

19.3 LuaJIT

LuaJIT 是 Lua 语言的另一个（非官方）实现，包括一个汇编语言编写的解释器和一个 JIT 编译器。前者使用的是汇编语言，速度比 Lua 官方的解释器还要快一点，而后者可以把 Lua 语言用 “Just-In-Time” 技术直接编译为目标机器码，使运行速度成倍提升，达到或接近 C/C++ 的程度。

^① Lua 解释器在启动时自动把标准库预加载到全局名字空间，所以标准库不需要使用 `require` 加载。

^② 本书作者习惯在定义模块时使用名字 “proto” 作为表的名字，但 OpenResty/Lua 的命名惯例是使用 “_M”，不采用惯例的原因是作者比较偏好小写化的名字。

LuaJIT 基于 Lua 5.1，但在不破坏兼容性的前提下适当引入了一些 5.2 和 5.3 的语言特性，还提供了很多特别的优化和库，例如 `table.new`、`bit`、`ffi` 等，所以比原生的 Lua 更加强大。^①

LuaJIT 是开源的，在官网 <http://luajit.org/> 上可以免费下载。当前的最新版本虽然是 beta 状态，但其实也很稳定，可以放心地用在生产环境。

19.3.1 continue

Lua 语言明确地不支持 `continue`，这让很多程序员都非常不适应，早期的解决办法是在循环里使用一个大的 `if-else`，非常不方便。好在 LuaJIT 扩展了 Lua 5.1 的语法，加入了 Lua 5.2 里的 `goto` 语句，可以变通地实现 `continue`。

`goto` 语句需要配合标签 “`::label::`” 使用，跳转到代码里的任何位置，非常灵活，但随意的跳转也很容易打乱程序的执行流程，导致难以理解维护的代码，Lua 里应当慎用。

在 `for/while/until` 循环里，代码块的最末尾加上一个 “`::continue::`” 的 label，就可以使用 “`goto continue`” 的方式跳过循环体里后续的代码，重新从开头执行循环，从而达成了 `continue` 的效果，例如：

```
for i=1,10 do                                -- for 循环语句开始
    if i % 2 == 0 then                         -- 检查是否是偶数
        goto continue                         -- goto 实现 continue，跳过后续代码
    end
    print("i = ", i)                          -- 打印数字
::continue::                                 -- 必须要在循环体末尾设置 label
end
```

跳转的标签 “`::continue::`” 名字不是固定的，也可以改用任意的标识符，但通常来说 `continue` 的名字最好，含义十分清晰。

19.3.2 bit

LuaJIT 为 Lua 5.1 增加了对整数类型的位运算能力，使 Lua 能够处理二进制数据。

LuaJIT 的位运算功能包括 `band`、`bor`、`bnot`、`bxor`、`lshift`、`rshift` 等函数，虽然没有 C/C++ 的操作符 `&`、`|`、`!`、`~`、`<<`、`>>` 等那么方便，但小心仔细地编写代码也可以实现相同的功能。

^① 详细的 LuaJIT 扩展可参见 <http://luajit.org/extensions.html>。

模块 `bit` 提供位运算功能，需要使用 `require` 函数引入才能使用。

示范位运算的代码如下：

```
local bit = require "bit"                -- 加载 LuaJIT 的 bit 库

local band, bor = bit.band, bit.bor      -- local 别名简化使用
local bxor, bnot = bit.bxor, bit.bnot    -- local 别名简化使用
local lshift     = bit.lshift            -- local 别名简化使用
local tohex      = bit.tohex             -- tohex 把数字转换为十六进制字符串

local x, y = 1, 4                        -- 两个整数变量
print(band(x, y))                       -- 位与操作, 1&4=0
print(bor(x, y))                        -- 位或操作, 1|4=5
print(bxor(x, y))                       -- 异或操作, 1^4=5
print(tohex(bnot(x)))                   -- 位非操作, ~1=fffffffe
print(lshift(y, 1))                     -- 左移操作, 4<<1=8
```

使用 `bit` 库配合 `string.byte` 可以较容易地解析二进制数据，例如常见的 4 字节整数：

```
local byte = string.byte                 -- string 库里取字节的函数
local data = '\000\000\001\001'         -- 一个二进制数据

local a,b,c,d = byte(data, 1, 4)        -- 取四个字节

local x = blshift(a, 24) +               -- 第一个字节左移 24 位
          blshift(b, 16) +               -- 第二个字节左移 16 位
          blshift(c, 8) +                -- 第三个字节左移 8 位
          d                               -- 最后一个字节不需要左移
print(x)                                -- 位运算的结果是 257
```

19.3.3 ffi

`ffi` 是 LuaJIT 里最有价值的一个库，它极大地简化了在 Lua 代码里调用 C/C++ 函数的工作，而且执行效率更高，技术上完全超越了 Python 和 Java (JNI)。

`ffi` 库不需要编写繁琐的 Lua/C 绑定函数，只要在 Lua 代码里嵌入 C 函数或数据结构的声明，不必编写额外的代码即可直接调用 C 接口，非常方便。

`ffi` 库不仅可以调用系统函数和 OpenResty/Nginx 内部的 C 函数，还可以加载 `so` 形式的动态库，调用动态库里的函数，从而轻松灵活地扩展 Lua 的功能。

`ffi` 库同样需要使用 `require` 引入后才能使用，较常用的功能有：

■ `ffi.load` : 加载 *.so 动态库；

- `ffi.null` : 相当于 C 语言里的 `NULL`;
- `ffi.string` : 把 C 指针指向的内存数据转换为 Lua 字符串;
- `ffi.cdef` : 声明 C 函数或数据结构, 通常使用 “[...]” 的形式;
- `ffi.typeof` : 创建一个 C 结构 “类型对象”;
- `ffi.new` : 使用 C “类型对象” 分配可供 C 函数操作的内存;
- `ffi.C` : C 函数所在的表, 可以用点号 “.” 访问并直接执行。

`ffi` 的工作机制比较复杂, 本书仅使用一个小例子来示范它的用法, 下面的代码调用了 UNIX 系统函数 `gettimeofday()`, 获取微秒精度的时间:

```
local ffi = require "ffi" -- 加载 LuaJIT 的 ffi 库

local ffi_null = ffi.null -- local 别名简化使用
local ffi_cdef = ffi.cdef -- local 别名简化使用
local ffi_new = ffi.new -- local 别名简化使用
local ffi_C = ffi.C -- local 别名简化使用

ffi_cdef[[
struct timeval { -- C 代码里必须使用 C 风格的注释
    long int tv_sec; -- 这种风格在 Lua 里久违了 (笑)
    long int tv_usec;
};
int gettimeofday(struct timeval *tv, void *tz); // C 函数声明
]]

local timeval_t = ffi_typeof("struct timeval") -- 产生一个 C 结构的 “类型对象”
local tm = ffi_new(timeval_t) -- 分配供 C 函数使用的内存

ffi_C.gettimeofday(tm, ffi_null) -- 直接调用之前声明的 C 函数!

print(type(tm.tv_sec)) -- 查看数据类型, 是 cdata
print("sec:", tm.tv_sec) -- 直接输出 C 数据
print("sec:", tonumber(tm.tv_sec)) -- 在 Lua 里使用需要转换
```

这段代码里我们用 `ffi.cdef` 声明了 C 函数 `gettimeofday()` 和它的参数类型, 然后依次调用 `ffi.typeof` 和 `ffi.new` 分配内存空间, 最后用 `ffi.C` 执行了之前声明的 C 函数, 整个流程与写 C 代码非常相似, 但实际上却是 100% 的纯 Lua 程序。

注意从 C 函数里返回的数据类型, 是特殊的 `cdata`, 而不是 `number` (虽然也是数字), 不能直接在 Lua 里使用, 必须要转换之后才行。

`ffi` 库的功能非常强大, OpenResty 也专门为它做了很多的优化, 感兴趣的读者可参考网上的资料进一步学习。

19.4 Lua 模块

OpenResty 的两个核心组件是 Nginx 和 Lua/LuaJIT，而在这两个原本毫无关系的世界之间架起沟通桥梁的就是 Lua 模块，通常简称为 `ngx_lua`。

`ngx_lua` 的全名是 `ngx_http_lua_module`，是工作在 Nginx HTTP 框架里的一个模块，基于 Nginx 的 HTTP 机制处理客户端请求。但它又不是一个简单的 HTTP 模块，而是糅合了 `handler/filter/upstream/balance/subrequest` 等多种功能，可以在 Nginx HTTP 处理流程的 `init/rewrite/access/content/filter/log` 等多个 Phase 里执行 Lua 代码，并且能够用非常简单的形式访问几乎所有的 Nginx 环境数据，诸如配置、变量、`ctx`、请求参数、共享内存等，可以说是个“万能”的模块。

使用 `ngx_lua`，原本用 C 语言几百行才能实现的功能也许只需要几行 Lua 代码就能完成，真正实现了“一句顶一万句”。^①

19.4.1 指令简介

虽然 `ngx_lua` 很强大，但它本质上仍是一个 Nginx http 模块，所以也提供指令来调整模块的行为。

`ngx_lua` 包含的指令很多，目前约有 60 多个，但大体上可以分为两类：配置指令和功能指令。

配置指令与普通的 Nginx 模块指令类似，通常是一些开关、数值或者字符串，用来设置 `ngx_lua` 运行的基本参数，例如连接超时、定时器数量限制、Lua 脚本的查找路径等，比较容易理解使用。

功能指令与 Nginx 的 HTTP 运行机制密切相关，可以在各个阶段执行 Lua 代码，处理 HTTP 请求，每一个指令就相当于一个 `handler/filter` 模块的功能——但我们无须编写 C 代码，而是用 Lua 来实现业务逻辑。

本书不可能介绍所有的 `ngx_lua` 指令，完整详细的说明请参见官网。

19.4.2 配置指令

`ngx_lua` 的配置指令约有 30 个，较常用的指令有：

^① `ngx_lua` 的 GitHub 地址是“<https://github.com/openresty/lua-nginx-module/>”，说明文档非常详尽，建议读者随时参考。

- `lua_package_path` : 设置 Lua 脚本的查找路径;
- `lua_package_cpath` : 设置 Lua C 模块 (*.so) 的查找路径;
- `lua_code_cache` : 启用 Lua 代码缓存, 加速执行速度;
- `lua_malloc_trim` : 定期清理内存, 减少内存的占用;
- `lua_shared_dict` : 定义一块共享内存;
- `lua_check_client_abort` : 检测客户端断连。

前两个指令是 `ngx_lua` 运行的关键, 以字符串的形式确定 Lua 脚本和库的查找路径, 多个路径使用 “;” 分隔, 默认的查找路径用 “;;”, 例如:

```
lua_package_path    "/usr/local/openresty/lualib/?.lua;;";
lua_package_cpath   "/usr/local/openresty/lualib/?.so;;";
```

这两条指令告诉 `ngx_lua` 在 OpenResty 的标准安装目录里查找 Lua 脚本文件和 C 模块。

`lua_package_path`/`lua_package_cpath` 指令里还可以使用特殊变量 “\$prefix”, 表示 Nginx 启动时的工作路径 (即 “-p” 参数指定的目录), 这样就可以不使用绝对路径, 配置更加灵活。

我们应该充分利用 `lua_package_path`/`lua_package_cpath` 指令支持多查找路径的功能, 分目录存放不同功能的 Lua 脚本, 避免在一个目录 (尤其是 OpenResty 的安装目录) 里存放大量的业务代码。

本书使用的查找路径配置是:

```
lua_package_path
    "/usr/local/openresty/lualib/?.lua;$prefix/service/?.lua;;";
```

这样, 我们的自有代码和库都存放在了工作目录的 `service` 目录里, 而 OpenResty 标准库的位置不变, `ngx_lua` 会优先在 `service` 目录里查找库。^①

19.4.3 功能指令

如果读者对 Nginx 的 HTTP 处理流程有所了解, 那么就会发现 `ngx_lua` 的功能指令基本上与各个处理阶段对应, 可以把 Lua 代码插入进 Nginx 定义的这些阶段, 执行业务逻辑。

常用的 `ngx_lua` 功能指令有:

- `init_by_lua` : 在 `postconfiguration` 阶段执行 Lua 代码;

^① 把 Lua 脚本放置在 `conf` 目录里不是个好主意, 因为 Lua 代码并不是配置, 而是 Web 服务程序。

- `init_worker_by_lua` : 在 `init_process` 阶段执行 Lua 代码;
- `set_by_lua` : 在 `rewrite` 阶段操作 Nginx 变量;
- `rewrite_by_lua` : 在 `rewrite` 阶段执行 Lua 代码;
- `access_by_lua` : 在 `access` 阶段执行 Lua 代码;
- `content_by_lua` : 在 `content` 阶段执行 Lua 代码, 产生响应内容;
- `header_filter_by_lua` : 在 `header_filter` 阶段执行 Lua 代码;
- `body_filter_by_lua` : 在 `body_filter` 阶段执行 Lua 代码;
- `log_by_lua` : 在 `log` 阶段执行 Lua 代码;
- `balancer_by_lua` : 在负载均衡阶段执行 Lua 代码。

可以看到, `ngx_lua` 几乎为所有 Nginx 可用的阶段都定义了相应的 `xxx_by_lua` 指令, 指令的名字含义很清晰, 可以望文生义。

一个简单的功能指令的例子如下:

```
content_by_lua_block {
    ngx.say("hello openresty")
}
```

-- content 阶段, Lua 产生响应内容
-- 向客户端输出一个字符串

`ngx_lua` 的功能指令通常都有三种形式 (少数新指令例外):

- `xxx_by_lua` : 执行指令后字符串形式的 Lua 代码;
- `xxx_by_lua_block` : 功能相同, 但指令后是 `{...}` 的 Lua 代码块;
- `xxx_by_lua_file` : 功能相同, 但执行磁盘上的脚本文件。

`xxx_by_lua` 是 `ngx_lua` 早期的指令形式, Lua 代码长度受 Nginx 配置文件的限制 (不能超过 4K), 而且在字符串里要考虑单引号和双引号的冲突, 编写代码不太方便, 不推荐使用。

`xxx_by_lua_block` 是它的改进形式, 可以在花括号 “`{}`” 里书写任意的 Lua 代码, 没有转义符、单引号和双引号的限制 (得益于 Nginx 优秀的配置文件解析设计, `ngx_lua` 自定义了配置块的解析函数)。

`xxx_by_lua_file` 是最推荐使用的方式, 它彻底分离了 Nginx 配置文件与 Lua 代码, 让两者可以独立部署, 而且文件形式也更容易以模块的方式组织 Lua 程序。

指令里的 Lua 脚本文件可以用绝对路径或者相对路径来指定, 如果使用相对路径, 那么查找路径是 Nginx 启动时的工作路径 (即 “`-p`” 参数指定的目录), 注意它与 `lua_package_path`/`lua_package_cpath` 指令没有任何关系 (虽然可能都是一个目录), 例如:

```
content_by_lua_file service/http/xxx.lua; #执行 service/http 里的 Lua 脚本
```

19.4.4 指令关系图

ngx_lua 的指令运行在 Nginx 的不同处理阶段, 图 19-3 表示了它们所在的阶段和执行的先后顺序。

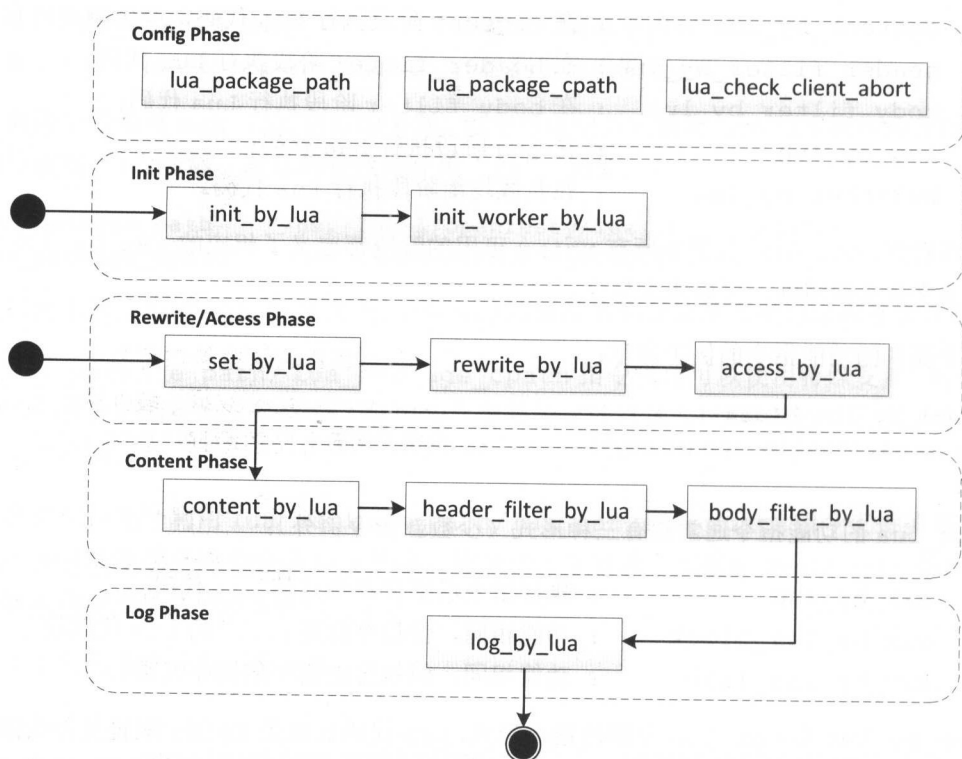


图 19-3 ngx_lua 的指令关系

19.4.5 应用开发流程

使用 OpenResty/ngx_lua 开发 Web 应用的流程与开发 Nginx 模块不太一样, 没有编译, 基本步骤是: ①

- 1) 确定对外服务的端口 (通常是 80), 也就是定义 Nginx 的 server;
- 2) 确定对外服务的 URI, 也就是定义 Nginx 的 location;
- 3) 确定执行业务逻辑的一个或多个具体阶段, 如 rewrite/access/content 等;

① 但有特殊需求也可以把 Lua 脚本编译成字节码, 甚至直接静态链接进 Nginx 可执行文件。

- 4) 在 location 里使用 ngx_lua 指令，配置基本参数和要执行的脚本；
- 5) 根据需求编写一个或多个 Lua 脚本，实现业务；
- 6) 调试、测试，最后部署上线运行。

可见，开发 OpenResty 应用主要的工作有两个，分别是编写配置文件和编写 Lua 代码，前者决定了 Lua 代码运行的位置和时机，后者决定了具体的功能实现。

本书在 conf 目录里存放 Nginx 配置文件，在 service 目录里存放 Lua 功能代码，具体目录结构见 19.1.6 节。

19.5 功能接口

ngx_lua 基于 Nginx 框架为用户提供了上百个功能接口，这些接口都是纯 Lua 的，完全屏蔽了底层的 Nginx C 数据结构细节，不需要关心内存池、动态数组、缓冲区、数据块链、红黑树等概念，ngx_lua 会自动完成 Lua/Nginx 两个世界之间的数据“翻译工作”。

ngx_lua 的所有功能都位于表 ngx 里，可全局访问，不需要 require。

受 Nginx 框架自身的约束，有的 ngx_lua 函数在某些功能指令里是不能使用的，例如 cosocket (19.5.6 节) 就不能在 set_by_lua 里使用，但存在绕过这些限制的变通办法。

本节的示例代码均使用指令“content_by_lua_file”，Nginx 配置如下：^①

```
location ~ ^/(\w+) {                                #使用正则表达式定义 location
    content_by_lua_file service/http/$1.lua;        #使用 uri 作为 Lua 脚本名
}
```

当执行“curl 127.0.0.1/xxx”时，OpenResty 就会执行 service/http 目录里对应的 Lua 脚本。

19.5.1 运行日志

函数 ngx.log(log_level, ...) 记录 Nginx 运行日志 (logs/errors.log)，与 Nginx 的 ngx_log_error() 对应，用法很类似 Lua 的 print()，可以接受任意多个参数，记录任意信息。

函数的第一个参数是日志级别，取值可以是 ngx.INFO/ngx.NOTICE/ngx.WARN/ngx.ERR 等，用法示例如下：

^① 这只是本书为了方便做的配置，实际开发中需要对 uri 做检查，避免执行系统里的其他文件。

```

ngx.log(ngx.INFO, "hello openresty")           -- INFO 级别的日志
ngx.log(ngx.WARN, "warning is ", "some messgae") -- 传递多个参数记录日志
ngx.log(ngx.DEBUG, debug.traceback())          -- 也可以用 debug 库

```

由于 Nginx 对日志文本长度有限制，上限约 2K 个字符，所以不能在日志里记录大段的文字，超长的字符串会被截断。

在记录日志时应当尽量少用字符串连接操作“..”，不仅可以节约内存，而且还避免了参数为 nil 时无法连接的错误。

19.5.2 时间与日期

ngx_lua 提供下列的时间日期函数：

- ngx.sleep : 同步非阻塞的睡眠函数，可用来暂停一小段时间；
- ngx.time : 返回当前的时间戳，即 epoch 以来的秒数；
- ngx.now : 类似 ngx.time，但返回是个浮点数，精确到毫秒；
- ngx.today : 当天的日期，格式是 yyyy-mm-dd；
- ngx.localtime : 获取本地时间，格式是 yyyy-mm-dd hh:mm:ss；
- ngx.utctime : 获取 UTC 时间，格式是 yyyy-mm-dd hh:mm:ss；
- ngx.cookie_time : 把时间戳转换为 cookie 时间格式；
- ngx.http_time : 把时间戳转换为 http 时间格式；
- ngx.update_time : 强制更新时间，可获得更准确的时间，但成本较高；
- ngx.parse_http_time : 解析 http 时间格式，转换为时间戳。

下面的代码示范了部分函数的用法：

```

local secs    = ngx.time()           -- 取当前时间戳
local msec    = ngx.now()            -- 取当前毫秒精度的时间戳
assert(msecs - secs < 1)             -- now() 比 time() 多了小数部分

print(ngx.http_time(secs))           -- 转换为 http 时间格式

```

在实践中我们通常使用 ngx.now 获取更精确的时间用来计时，但要注意的是虽然它返回的是个浮点数，精确到小数点后的若干位，但并没有实际意义，因为 Nginx 自身的时间精确度就是毫秒。

想要获取更高精确度的时间可以通过 ffi 库调用系统函数 gettimeofday()，参见 19.3.4 节。

19.5.3 变量

在 OpenResty/nginx_lua 里使用 Nginx 变量非常容易, 表 ngx.var 里包含了所有的 Nginx 内置变量和自定义变量, 可以用 “[]” 或者 “.” 任意访问。如果变量是可写的 (通常由 set 指令定义), 还可以直接修改。

ngx.var 的用法示范如下:

```
print(ngx.var.uri)           -- 输出变量$uri, 请求的 uri
print(ngx.var['http_host'])  -- 输出变量$http_host, 请求头的 host

if #ngx.var.is_args > 0 then  -- 检查是否有 uri 参数, $is_args
    print(ngx.var.args)       -- 输出 uri 参数, $args
end
```

注意在检查是否有 uri 参数时必须用 “#ngx.var.is_args > 0” 的方式, 这是因为 \$is_args 是字符串, 不是 nil 或 false, 在 Lua 里判断空字符串只能用检查长度的方式。

利用好 ngx.var 能够获取 Nginx 里的很多信息, 例如请求地址、请求参数、请求头、客户端地址、收发字节数等, 可以在 access_by_lua 里实现访问控制。

19.5.4 正则表达式

ngx_lua 在表 ngx.re 里提供五个正则表达式相关函数, 它们的底层实现是 Nginx 的 PCRE 库, 并且支持 PCRE-JIT, 速度非常快, 完全可以代替 Lua 标准库的字符串匹配函数。

这些正则表达式函数包括 (篇幅所限, 详细接口参数无法列出): ^①

- ngx.re.match : 正则匹配, 同时捕获子表达式;
- ngx.re.find : 正则查找, 返回查找到的位置索引;
- ngx.re.gmatch : 多次正则匹配 (以迭代器的方式);
- ngx.re.sub : 正则替换;
- ngx.re.gsub : 多次正则替换。

在使用这些函数时应当尽量使用 “[...]” 的方式书写正则表达式, 可以避免转义符带来的困扰, 并使用 “jo” 参数启用 PCRE JIT 编译, 加快正则表达式的处理速度。

下面的代码简单示范了其中几个函数的用法:

^① OpenResty 在库 lua-resty-core 里提供非常有用的正则字符串切分函数 ngx.re.split。

```

local str = "abcd-123"                -- 一个字符串
str = ngx.re.sub(                      -- 正则替换, "jo" 参数启用 PCRE JIT 编译
    str, "ab", "cd", "jo")            -- 把 ab 替换成 cd
assert(str == "cdcd-123")              -- 替换的结果是 "cdcd-123"

local m = ngx.re.match(                -- 正则匹配, 返回一个匹配结果表
    str, "(.*)123$", "jo")            -- 匹配 "123" 结尾的字符串
assert(m and m[1] == "cdcd-")         -- 匹配成功, 子表达式存储在表里

local from, to = ngx.re.find(          -- 正则查找
    str, [[\d+]], "jo")               -- 查找数字
assert(string.sub(str, from) == "123") -- 使用返回的索引位置取子串

```

函数 `ngx.re.match()` 返回的结果是一个表, 里面存储了匹配的结果。如果匹配成功, `m[0]` 保存的是整个 (匹配成功的) 字符串, 之后的 `m[1]`、`m[2]` 等保存的是匹配的子表达式。

19.5.5 请求处理

OpenResty/nginx_lua 对 HTTP 请求的处理提供非常完善的支持, 大约有 40 个接口, 可以操作请求行、请求头、请求体、响应头、响应体, 执行重定向、跳转等, 而且因为 Lua 内建支持协程, 不需要像 C 代码那样编写异步回调的 handler, 能够以同步的方式编写非阻塞代码, 非常方便自然。

OpenResty/nginx_lua 里 HTTP 处理请求相关的接口较多, 本节只能对它们做简要介绍, 详细的说明请参见官网。

环境数据

表 `ngx.ctx` 是 HTTP 请求的“环境数据”, 相当于 Nginx 里的 `r->ctx`。它是每个请求所独有的, 可以用来存储一些临时的数据, 在整个处理流程中共享。

但 `ngx.ctx` 使用的成本较高, 应当尽量少用, 只存放少量必要的的数据, 避免滥用。

请求行

HTTP 请求行的信息包括请求方法、URI、HTTP 版本等, 这些虽然可以用 `ngx.var` 以变量的方式获取, 但效率不高, 而且是只读的, 所以 OpenResty/nginx_lua 在表 `ngx.req` 里提供了一些专门操作请求行的函数, 包括:

- `start_time` : 请求的开始时间, 相当于 `$request_time`;
- `get_method` : 获取字符串形式的请求方法;
- `set_method` : 改写请求方法, 使用数字常量, 不能用字符串;

- `set_uri` : 改写 URI;
- `get_uri_args` : 获取 URI 的参数, 以表的形式返回解析后的结果;
- `get_post_args` : 获取 POST 方式传递的参数, 同样返回一个表;
- `set_uri_args` : 改写 URI 参数。

这些函数可以用在 `rewrite` 阶段, 改写 URI 的各种参数, 实现跳转。

请求头

HTTP 请求头都是 Key-Value 的形式, 很适合用 Lua 里的表来存储, 处理请求头的接口也位于表 `ngx.req` 里, 包括:

- `raw_header` : 获取请求头的未解析字符串形式 (含请求行);
- `get_headers` : 获取所有的 HTTP 请求头, 返回一个表;
- `set_header` : 改写一个 HTTP 请求头;
- `clear_header` : 删除一个 HTTP 请求头。

后两个函数通常用在 `rewrite` 阶段, 改写 HTTP 请求。

请求体

OpenResty/`ngx_lua` 在表 `ngx.req` 里提供的处理请求体的函数如下:

- `discard_body` : 丢弃请求体, 即忽略请求头后的数据;
- `read_body` : 开始读取请求体数据;
- `get_body_data` : 获取请求体数据, 是非阻塞的;
- `set_body_data` : 改写请求体数据;
- `get_body_file` : 如果请求体很大, 存储在文件里, 返回使用的文件名;
- `set_body_file` : 把请求体数据写入文件, 文件必须已经存在;
- `init_body` : 创建一个新的请求体, 替代原请求的数据;
- `append_body` : 向 `init_body` 创建的请求体里添加数据;
- `finish_body` : 完成请求体数据的创建。

响应头

OpenResty/`ngx_lua` 能够处理 HTTP 响应里的状态码和头信息 (不能改状态行):

- `ngx.status` : 读写响应状态码;
- `ngx.header` : 读写响应头;
- `ngx.send_headers` : 发送响应头, 是非阻塞的;

- `ngx.headers_sent` : 检查响应头是否已经发送。

`ngx.header` 以表的方式操作 HTTP 响应头, 可以用 “[xxx]” 的形式直接访问, 也可以用点号 “.” 以字段的形式访问, 后一种形式里要用 “_” 来代替头名字里的 “-”, 例如:

```
ngx.header.content_length = len          -- 相当于 header['Content-Length']
```

响应体

在 OpenResty/ngx_lua 里处理响应体很简单, 不需要考虑缓冲、异步、回调等问题:

- `ngx.print` : 发送数据, 支持多参数, 参数可以是字符串、数字或者表;
- `ngx.say` : 同 `print`, 但在数据的最后会加一个换行符, 即 ‘\r’;
- `ngx.flush` : 刷新缓冲发送数据。

这三个函数都是非阻塞的, OpenResty 内部会使用 Lua 协程自动处理数据的发送, 即使是很大的数据 (几 M 或几十 M) 也不会阻塞整个 Nginx 进程。^①

流程控制

OpenResty/ngx_lua 里还有几个特别的函数用来控制 HTTP 处理流程:

- `ngx.exit` : 立即结束请求的处理, 返回的状态码可以用参数指定;
- `ngx.eof` : 结束发送响应数据, 但可以继续执行后面的处理逻辑;
- `ngx.exec` : 执行 server 内部的 location 跳转;
- `ngx.redirect` : 执行标准的 301/302 重定向。

其中 `ngx.eof` 比较常用, 可以尽早返回给客户端响应数据, 然后再执行统计、日志、存储等收尾工作, 减少客户端感知的等待时间。

19.5.6 请求转发

OpenResty/ngx_lua 提供强大的 `cosocket` 功能, 它源自 Nginx 的 `upstream` 机制, 但又超越了 `upstream` 机制, 结合了 Nginx 的事件机制和 Lua 的协程特性, 以同步非阻塞的方式实现 `socket` 编程, 高效地与任意的后端服务通信。^②

`cosocket` 的接口与 `socket C API`/Lua API 类似, 学习的成本很低, 能够在很短的

① `ngx.flush` 提供额外的同步等待操作 (也是非阻塞的), 参数 `true` 可以强制等待数据发送成功。
② `cosocket` 是 OpenResty 独有的概念, 正式的名字是 “coroutine based socket”, 已经得到了 agentzh 本人的确认。

时间里掌握，任何人都可以很轻松地编写出高质量高性能的网络通信程序。

`cosocket` 还内建连接池（基于 Nginx 的连接池机制），可以实现重要的长连接功能，解决了网络编程领域里的一大难题。

`cosocket` 支持 TCP、UDP 和 UNIX Domain Socket，三者的接口基本相同，故本书只介绍使用 TCP 协议的 `cosocket`。

函数 `ngx.socket.tcp()` 创建一个 TCP 协议的 `cosocket` 对象，能够与任意的 TCP 服务器通信，可以使用的方法有：

- `setTimeout` : 设置 TCP 通信的超时时间；
- `settimeouts` : 分别设置建立连接/读/写的超时时间；
- `connect` : 建立与后端服务器的连接；
- `sslhandshake` : 发起 SSL/TLS 握手；
- `send` : 发送数据；
- `receive` : 接收数据；
- `receiveuntil` : 以指定的模式持续接收数据；
- `close` : 关闭连接；
- `setkeepalive` : 把连接放入连接池，即长连接复用；
- `getreusedtimes` : 获取连接的复用次数。

注意这些操作都是完全非阻塞的（100% nonblocking），不需要编写回调函数就可以实现高性能的并发编程。

因为 TCP 协议应用的最多，所以 OpenResty/nginx_lua 还提供了一个简化操作：`ngx.socket.connect()`，它相当于 `ngx.socket.tcp()` 后紧接着调用 `connect()`，返回的是一个已经成功创建连接的 `cosocket` 对象（TCP 协议）。

使用 `cosocket` 可以很容易地连接一个或多个后端服务器，无阻塞地收发数据，再利用 `bit/ffi` 库解析二进制数据，就能够实现 Nginx upstream C 模块的功能，而且更加灵活。

`cosocket` 只能连接到一个确定的地址，不能利用 Nginx 的 load-balance 机制，但这个问题也不难解决，只要在 Lua 代码里用表来保存后端服务器集群，然后自己实现负载均衡算法就可以了。^①

^① OpenResty 里另有一个模块 `ngx_http_lua_upstream`，可以在 Lua 代码里获取 Nginx 配置里定义的 upstream 集群，然后再做负载均衡策略。

19.5.7 子请求

OpenResty/nginx_lua 可以发起一个或多个 Nginx HTTP 子请求，也完全是非阻塞的。

函数 `ngx.location.capture(uri, {...})` “调用”一个本 server 内的 location，第二个参数用来传递请求的 `method`、`args`、`body`、`ctx`、`vars` 等额外数据。当子请求执行完毕后，函数返回一个表，可以用 `status`、`header`、`body` 等字段获得响应数据。

通常发起子请求的形式是：

```
local res = ngx.location.capture(uri,      -- 发起一个子请求
                                {method = ngx.HTTP_POST,  -- 修改子请求的方法，可以改成 POST
                                  args = {...},           -- 以表的形式修改子请求的参数
                                  body = ...              -- 也可以添加请求体数据
                                })              -- 发起后同步非阻塞等待子请求执行完毕

if res.status == ngx.HTTP_OK then          -- 检查子请求的状态码
    ngx.print(res.body)                    -- 获取响应体数据，任意处理
end
```

虽然在 OpenResty/nginx_lua 里发起子请求比 C 代码要简单的多，但自 1.10 开始，Nginx 下调了子请求并发的上限，主请求最多只能并发 50 个子请求，相当于发出了一个明确的信号：不推荐使用子请求功能。

因此，本书作者建议尽量不使用 `ngx.location.capture`，而是改用 `cosocket`，它完全位于 OpenResty/nginx_lua 的世界里，更灵活可控。

19.5.8 定时器

函数 `ngx.timer.at` 创建一个定时器，当时间到时执行一个 Lua 函数，可以用它来延后某些费时的操作，或者绕过 `init_by_lua`、`set_by_lua`、`log_by_lua`、`header_filter_by_lua` 等指令的限制，运行任意的 Lua 代码。

`ngx.timer.at` 的形式是：

```
local function handler(premature, ...)      -- 定时器的回调函数
    ...                                     -- 里面可以是任意的 Lua 代码
end                                          -- 回调函数定义结束
ok, err = ngx.timer.at(delay, handler, ...) -- 启动定时器，delay 秒后到期
```

函数 `handler` 是定时器要执行的函数，它使用的参数可以从 `ngx.timer.at` 传递，但第一个参数必须是 `premature`，函数可以由此判断 Nginx worker 进程是否正在退出 (`exiting`)，执行必要的清理动作。

定时器 handler 的执行是与 HTTP 请求处理分离的，所以在函数里不能使用 `ngx.var`、`ngx.req` 和 `ngx.print` 等请求相关的函数，但其他的大多数功能都是可用的，比如 `cosocket`、共享内存等。

定时器的时间参数 `delay` 单位是秒而不是毫秒，可以用小数形式指定毫秒级别的时间，如果 `delay=0`，那么定时器将没有延迟，会立即执行 handler 函数。

19.5.9 共享内存

`ngx_lua` 的指令 `lua_shared_dict` 定义共享内存，格式是：

```
lua_shared_dict shm_name size;           #定义一个大小为 size 名为 shm_name 的共享内存
```

它只能在 `http{}` 里使用，不能出现在 `server{}` 或 `location{}` 里，例如：

```
lua_shared_dict stats 32m;               #定义一个 32m 的共享内存，名字是 stats
```

`lua_shared_dict` 指令定义的共享内存表示为 `ngx.shared` 表里的一个对象，可以用与 Redis 命令类似的方法来操作：

- `get` : 从共享内存里获取一个值；
- `set` : 向共享内存里写入一个值，可以设置过期时间；
- `add` : 类似 `set`，但只有 `key` 不存在时才写入；
- `replace` : 与 `add` 相反，只有 `key` 存在时才写入；
- `delete` : 删除共享内存里的一个 `key`；
- `incr` : 原子递增整数；
- `lpush/rpush` : 队列操作，在两端添加数据；
- `lpop/rpop` : 队列操作，在两端弹出数据；
- `llen` : 获得队列里的元素数量；
- `flush_all` : 清空共享内存里的所有数据；
- `flush_expired` : 清空共享内存里的所有过期数据。

`OpenResty/ngx_lua` 的这个共享内存功能非常有用，极大地便利了 `worker` 进程间的通信和协作，而且它还提供类似 Redis 的数据结构，可以当作一个简易“数据库”来使用，同时没有网络通信的成本，速度很快。^①

共享内存存在 `OpenResty` 里用途很广，基于它可以实现流量统计、缓存、进程锁等高级功能，简单的示范代码如下：

^① `agentzh` 有计划使用共享内存实现一个轻量级的关系数据库，进一步增强共享内存的实用性。

```

local stats = ngx.shared.stats           -- 获取之前定义的共享内存对象

stats:set("count", 1, 10)                -- 在共享内存里设置一个值, 10 秒后过期
local v = stats:get("count")             -- 获取共享内存里的值

local id = stats:incr("uid", 0, 1)        -- 原子增加一个整数
assert(id == 1)                          -- 检查增加后的整数值

stats:rpush("sessions", "xxx")           -- 队列操作, 在右端插入一个元素

v = stats:lpop("sessions")               -- 队列操作, 从左端取出一个元素
assert(v == "xxx")                       -- 检查从队列里取出的元素

stats:flush_all()                        -- 清空共享内存里的所有数据

```

注意在这段代码里我们对共享内存对象使用了“:”操作符, 这是 Lua 里对象操作的特殊用法 (可以近似地理解为 C/C++/PHP 里的箭头操作符“->”)。

19.6 应用实例

本节将基于之前介绍的 OpenResty/ngx_lua 指令和功能接口, 开发一系列 OpenResty 应用, 实现之前第 8 章~第 10 章里 C/C++ 模块的等价功能。

开发 OpenResty 应用不能使用 Nginx 指令来定义配置参数 (因为指令必须用 C 代码实现), 通常可以用下面的两种方式来配置 OpenResty 应用。

第一种方式是使用 Nginx 变量, 然后在 Lua 代码里用 ngx.var 来获取。这种方法简单方便, 但也有很多缺点。首先是变量的成本问题, 过多地使用变量会影响性能; 其次变量都是字符串形式, 对于数字值必须要调用 tonumber() 转换, 容易弄错; 最后, 变量的本意并不是用作配置的, 可能会被其他模块或 Lua 脚本修改, 存在安全隐患。

第二种方式是作者建议的方式, 就是用 Lua 代码定义配置, 数据直接就是 Lua 格式, 编写起来更加灵活, 并且很容易实现配置数据与逻辑代码的分离, 只要在 Lua 代码里用 require 函数加载配置脚本即可。

但为了便于讲解描述, 在本书中的示例没有实现分离, 配置数据与逻辑代码都写在了一个 Lua 脚本里, 实践中最好不要这样做。

19.6.1 处理请求

本节我们实现 8.10 节 ndg_echo_module 的功能。

基本设计是在“/echo”里实现请求处理，Lua 脚本是 echo.lua，配置如下：

```
location ~ /echo {                                     #Nginx 配置 location
    content_by_lua_file service/http/echo.lua;        #使用 Lua 产生响应内容
}
```

Lua 程序的逻辑与 8.10 节的 C++ 代码基本相同，但明显简单了很多：

```
local echo_string = "hello nginx\n"                  -- 配置发送给客户端的字符串

local method = ngx.req.get_method()                  -- 检查请求方法
if method ~= "GET" then                               -- 要求必须是 get
    return                                             -- 不是 get 则不处理直接退出
end

ngx.req.discard_body()                               -- 丢弃请求体

local args = ngx.var.args or ""                      -- 获取 uri 的参数

local len = #echo_string                             -- 计算字符串的长度
local str = echo_string                             -- 准备发送的数据

if #args > 0 then                                     -- 有参数则加上参数再发送
    len = #args + 1 + len                             -- 需要加上$args 的长度
    str = args .. "," .. str                         -- 参数字符串加在前面
end

ngx.header.content_length = len                      -- 设置响应体长度
ngx.status = 200                                     -- 设置状态码

ngx.send_headers()                                   -- 发送响应头，也可以不调用
ngx.print(str)                                       -- 发送响应体，会默认发送响应头
```

这段代码只有 20 余行，完全是业务逻辑，不需要编写额外的指令定义和模块定义，充分展示了 OpenResty 开发简易又强大的特性，读者可以对比一下 8.10 节，看看 Lua 代码是如何对应到 C/C++ 代码的。

19.6.2 过滤请求

本节实现 8.11 节 ndg_footer_module 的功能。

我们用两个 Lua 脚本分别实现 header_filter 和 body_filter 的功能，配置如下：

```
location ~ /echo {                                     #之前配置的 location
    content_by_lua_file service/http/echo.lua;
```

```
header_filter_by_lua_file service/http/header_filter.lua;
body_filter_by_lua_file   service/http/body_filter.lua;
}
```

header_filter 使用 ngx.header 操作响应头，加入新的信息：

```
local headers = {} -- 配置要新增的头信息
headers['x-name'] = 'chrono' -- 第一个头信息
headers['x-value'] = 'trigger' -- 第二个头信息

for k,v in pairs(headers) do -- 使用 pairs 函数遍历表
    ngx.header[k] = v -- 加入新的头信息
end

ngx.header.content_length = nil -- 删除长度信息
```

注意在过滤头时我们必须删除 Content-Length 头，因为 body_filter 会修改响应体数据，不能在这里确定实际的长度，这是与 C/C++ 模块不同的地方。

在 body_filter 里需要使用 ngx.arg[1] 和 ngx.arg[2] 来获取数据和 eof 信息，处理逻辑更简单：

```
local footer = "ocarina of time\n" -- 配置要添加的字符串

if not ngx.arg[2] then -- 不是最后一块数据
    return -- 不做任何处理
end

ngx.arg[1] = ngx.arg[1] .. footer -- 修改响应数据
```

19.6.3 转发请求

本节实现 9.6 节 ndg_upstream_module 的功能，配置如下：

```
location ~ /upstream { #Nginx 配置 location
    content_by_lua_file service/http/upstream.lua; #使用 Lua 访问后端服务器
}
```

OpenResty 应用需要使用 cosocket 连接后端服务器，用 LuaJIT 的 ffi 库解析返回的数据，为了节约篇幅，下面的代码里省略了对 cosocket 的错误处理：

```
local ffi = require "ffi" -- 加载 LuaJIT 的 ffi 库
local band, blshift = bit.band, bit.lshift -- 二进制位操作
local byte = string.byte -- 取字符串里的字节
```



```

local backend_addr = "127.0.0.1"      -- 配置连接的后端服务器地址
local backend_port = 2017             -- 配置连接的后端服务器端口

local args = ngx.var.args              -- 检查请求的参数
args = args and #args >= 4 and         -- 使用连续 and 运算
      string.sub(args, 1, 4) or "xxxx" -- 决定发送的字符串

local sock = ngx.socket.tcp()          -- 创建 cosocket 对象

local ok, err = sock:connect(          -- 非阻塞连接后端服务器
    backend_addr, backend_port)

local bytes, err = sock:send(args)     -- 非阻塞发送数据

local data, err = sock:receive(2)      -- 非阻塞接收响应头, 两个字节

local a,b = byte(data, 1, 2)           -- 分离两个字节
local len = blshift(a, 8) + b          -- 还原为表示长度的整数

data, err = sock:receive(len)          -- 非阻塞接收剩余的数据

ngx.say("received : ", data)           -- 把数据发给客户端

```

可以看到, 使用 OpenResty/ngx_lua 编写访问后端服务器的程序很简单, 是非常自然的同步方式——而且非阻塞, 没有难以理解的异步回调。

与共享内存对象一样, cosocket 对象也必须用“:”的特殊用法来调用 connect/send/receive 等方法。

19.6.4 子请求

本节实现 10.5 节 ndg_subrequest_module 的功能, 而 10.4 节的 ndg_data_hook_module 功能已经包含在了 OpenResty 里。

OpenResty 应用的配置如下:

```

location ~ /sub {                      #Nginx 配置 location
    content_by_lua_file service/http/sub.lua;  #使用 Lua 调用子请求
}

```

在 OpenResty 里使用子请求非常简单, 代码量几乎相当于零:

```

local uri = '/hello'                  -- 配置子请求的 uri
local args = 'chrono'                 -- 配置子请求的参数

```

```
local res = ngx.location.capture(          -- 发起子请求
    uri,                                  -- 传递子请求的 uri
    {args=args})                          -- 传递子请求的参数

ngx.say(res.status)                       -- 输出子请求的响应码
ngx.print(res.body)                       -- 输出子请求的响应数据
```

19.7 Stream Lua 模块

Nginx 从 1.9.0 版开始加入了 stream 模块, 支持处理 TCP/UDP 协议, 由于架构与 HTTP 类似, agentzh 也很快开发出了对应的 ngx_stream_lua_module, 通常简称为 stream_lua, 与 ngx_lua 类似, 可以让 Lua 代码在 Nginx Stream 框架里运行。

目前 stream_lua 还处于早期开发阶段, 达不到 ngx_lua 那样的完善程度, 缺少 access_by_lua、filter_by_lua 等功能指令 (不过也许读者拿到本书时会有所改善), 但由于与 ngx_lua 出自同样的基因, 已经是完全可用的状态了。例如, agentzh 就利用 stream_lua 实现了一个 Lua 版本的高效 DNS Server。^①

stream_lua 暂不包含在 OpenResty 发行版里, 需要我们在配置时使用 “--add-module=/path/to/stream-lua-nginx-module” 的方式重新编译 OpenResty。

19.7.1 功能接口

stream_lua 提供大部分 ngx_lua 里的功能接口, 名字与用法完全相同, 例如 ngx.log、ngx.re、ngx.now、ngx.ctx 等, 但因为它处理的是 TCP/UDP 协议而不是 HTTP 协议, 所以 ngx.req 里的 HTTP 相关函数都不存在, 只有一个重要的函数 ngx.req.socket, 用来获取表示客户端连接的 cosocket 对象, 用户需要使用它来完成 TCP/UDP 数据的收发处理。

还要注意一点: stream_lua 基于 Nginx 的 stream 框架, 与 ngx_lua 虽然很像但底层却是两个完全不同的运行环境, 使用彼此独立的 Lua VM 实例, 两者不能直接通信。

作为示范, 接下来的几个小节将使用 stream_lua 实现第 16 章里出现的几个应用协议。

^① 本书作者在 GitHub 上 fork 了一份 stream_lua 的代码, 地址是 <https://github.com/chronolaw/stream-lua-nginx-module>, 为它添加了 filter_by_lua、log_by_lua 等指令和 ngx.var 功能接口, 不过做的比较简单, 读者可参考。

19.7.2 discard

我们使用端口 781 提供 discard 协议，Lua 脚本是 discard.lua:

```
server {
    listen 781;
    content_by_lua_file service/stream/discard.lua;
}
```

discard 协议丢弃所有收到的数据，因为不能用 cosocket 接收无限长度的数据，所以我们每次只接收一个字节并丢弃（不必担心，不会对性能有任何影响）:

```
local sock = ngx.req.socket()           -- 获取客户端连接的 cosocket 对象

sock:settimeout(3000)                   -- 超时时间设置为 3 秒

local count = 0                          -- 统计收到的字节数

while true do                           -- 无限循环接收数据
    local data, err = sock:receive(1)    -- 一次接收 1 个字节，不会影响性能

    if not data or err then               -- 当客户端断连时结束循环
        break
    end

    count = count + 1                    -- 统计收到的字节数
    if count % 100 == 0 then              -- 把字节数记录到日志
        ngx.log(ngx.INFO, " received ", count, " bytes")
    end
end
```

19.7.3 time

time 协议服务端口是 782，Lua 脚本是 time.lua:

```
server {
    listen 782;
    content_by_lua_file service/stream/time.lua;
}
```

因为不需要读取客户端数据，所以 time.lua 非常简单，里面的代码只有一行:

```
ngx.say(ngx.time())                    -- 发送当前的时间戳，结束
```

19.7.4 chargen

chargen 协议使用端口 784, Lua 脚本是 chargen.lua:

```
server {
    listen 784;
    content_by_lua_file service/stream/chargen.lua;
}
```

chargen 协议也不需要读取客户端数据, 只要生成字符串再发送出去就可以了:

```
local count = 5; -- 配置发送数据的次数

local t = {} -- 使用表保存字符, 用于连接字符串
for i=0x20, (0x7f-1) do -- 循环生成 ASCII 码
    t[#t+1] = string.char(i) -- 把字符串插入表里
end

local str = table.concat(t) -- 连接表, 生成字符串
local len = #str; -- 计算字符串的长度

for i=1, count do -- 发送一定次数的字符串
    if i > len then -- i 是发送字符的偏移位置
        i = 1
    end
    ngx.say(string.sub(str, i), -- 把字符串拆成两部分发送
            string.sub(str, 1, i-1)) -- 实现字符序列的循环
    ngx.sleep(0.2) -- 发送后非阻塞睡眠 0.2 秒
end
```

使用 OpenResty 可以轻松地实现非阻塞睡眠功能, 而在 Nginx C 模块里则必须用 ctx 保存状态, 编写麻烦的回调函数才行。

19.7.5 echo

本节我们来实现 9.6 节和 19.6.3 节用到的后端 echo 服务, 端口是 2017:

```
server {
    listen 2017;
    content_by_lua_file service/stream/echo.lua;
}
```

echo.lua 里需要使用 ffi 库生成响应头里的长度字节, 实现如下 (省略了错误处理):

```
local ffi = require "ffi" -- 加载 LuaJIT 的 ffi 库
local band, brshift = bit.band, bit.rshift -- 二进制位操作
```

```

local tochar = string.char                -- 把整数转换为字符

local sock = ngx.req.socket()             -- 获取客户端连接的 cosocket 对象

local data, err = sock:receive(4)         -- 接收客户端发来的 4 字节数据

local timestamp = tostring(ngx.time())    -- 时间戳, 需转换为字符串形式
local len = #data + #timestamp            -- 计算响应数据的字节数

local header = tochar(brshift(band(len, 0xff00), 8)) -- 位运算得到字符串
    .. tochar(band(len, 0xff))

sock:send(header)                         -- 发送响应头数据, 可以与响应体合并
sock:send(data .. timestamp)              -- 发送响应体数据

```

19.8 lua-resty 库

在 ngx_lua 搭建的基础之上, agentzh 和众多 OpenResty 社区成员开发出了很多的 lua-resty 库, 覆盖了许多应用领域, 例如编码、压缩、密码学、SQL/NoSQL 数据库、消息中间件、图片处理、二维码等, 增强完善了 OpenResty 的生态圈, 组合利用这些库可以高效地开发出几乎任何类型的网络应用服务。

目前所有的 lua-resty 库都是基于 ngx_lua 实现的 (毕竟它出现的早), 但由于 stream_lua 具有与 ngx_lua 相同的功能接口, 所以大多数库也能够不加修改地直接运行在 stream_lua 里, 只有极少数的例外。

lua-resty 库数量众多, 这里只介绍三个较基本的库, 感兴趣的读者请参考 0.7 节的 GitHub 资源。

19.8.1 core

lua-resty-core 是 OpenResty 的一个官方库, 使用 LuaJIT 的 ffi 重新实现了 ngx_lua 里的很多功能, 效率比传统的 Lua 栈调用方式高很多, 此外它还提供一些 ngx_lua 没有的新功能, 强烈推荐使用。

OpenResty 自带了 lua-resty-core 库, 需要在启动时使用 init_by_lua 指令加载,

例如: ①

```
init_by_lua_block {
    require "resty.core"
    collectgarbage("collect")
}
```

-- 必须在 init 阶段初始化
-- 加载 resty.core 模块
-- 要求 Lua VM 回收清理内存

lua-resty-core 库的新增功能里包含一个非常有用的正则表达式函数 `split`, 可以简单地切分字符串, 但必须显式加载 `ngx.re` 模块才能使用, 用法示例如下:

```
local ngx_re = require "ngx.re"
local res, err =
    ngx_re.split("a,b,c,d", ",", "")

assert(res and #res == 4)
assert(res[1] == 'a' and res[4] == 'd')
```

-- 显式加载 ngx.re 模块
-- 使用 "," 切分字符串
-- 切分成功, 有 4 个结果
-- 检查切分的结果

lua-resty-core 库目前仅能用于 `ngx_lua`, 很遗憾不能在 `stream_lua` 里使用。

19.8.2 cJSON

lua-cjson 用于处理 JSON 格式的数据, 它的底层是 C 实现, 所以速度很快。②

lua-cjson 提供两个模块: “`cjson`” 和 “`cjson.safe`”, 我们通常使用后者, 顾名思义它更“安全”一些, 当数据格式错误时不会导致 Lua VM 错误, 而是返回 `nil`。

lua-cjson 用法比较简单, `encode` 对数据编码, `decode` 把数据解码为 Lua 表:

```
local cjson = require "cjson.safe"

local str = cjson.encode(
    {name='jojo', cat = 'comic'})

local obj = cjson.decode(str)

assert(obj.name == 'jojo')
assert(obj.cat == 'comic')
```

-- 加载 cjson.safe 模块
-- 把 Lua 表编码为 JSON 字符串
-- 解码字符串为 Lua 表
-- 检查解码的结果

① agentzh 计划在今后的版本里废弃 Lua C 函数栈调用的实现, 改用 LuaJIT 的 `ffi` 库完全实现所有的 `ngx_lua` 接口, 所以 lua-resty-core 库当前的加载方式可能会有所变动。

② 另外有一个 JSON 库实现 lua-resty-json, 性能比 lua-cjson 更好一些, 但并不包含在目前的 OpenResty 发行版里。

19.8.3 redis

Redis 是非常流行的内存 KV 存储系统，以速度快和丰富的数据类型而闻名，可以用在缓存、消息队列、数据库等领域，很多国内外知名公司都是它的用户。^①

lua-resty-redis 库基于 cosocket 实现了非阻塞的 Redis 客户端，支持 Redis 的所有命令和管道操作，默认包含在 OpenResty 发行包内，可以直接使用。

下面的代码简单示范了 lua-resty-redis 库的用法（省略了错误处理的代码）：

```
local redis = require "resty.redis"           -- 加载 resty.redis 模块

local rds = redis:new()                       -- 新建一个 redis 连接对象

ok, err = rds:connect("127.0.0.1", 6379)     -- 连接 Redis 服务器

ok, err = rds:set("metroid", "prime")        -- 向 Redis 写入数据

local res, err = rds:get("metroid")          -- 从 Redis 读取数据
assert(res == "prime")                       -- 检查读取的数据

rds:init_pipeline()                          -- 启动管道操作，加快执行速度
for i=1,10 do
    ok, err = rds:rpush('numbers', i)        -- 连续发送多个 Redis 命令
end                                           -- 向 Redis 队列里添加数据

results, err = rds:commit_pipeline()         -- 提交管道操作

rds:set_keepalive(10*1000, 100)             -- 加入连接池，长连接复用
```

在使用 Redis 时对于大批量的数据操作最好使用管道，它可以极大地压缩网络时延，加快处理速度。

如果 Redis 版本在 2.6 以上，还可以把复杂的交互处理过程编写成 Lua 脚本，再配合管道进一步提升性能。

19.9 总结

OpenResty 是一个“比 Nginx 更好的 Nginx”，它以 Nginx 为基础，集成众多设计精

^① Redis 2.6 之后加入了对 Lua 脚本的支持，所以可以在 OpenResty 的 Lua 代码里编写运行在 Redis 里的 Lua 代码，这种感觉“很奇妙”。

良的模块和工具，搭建出了完善易用的服务器开发环境，可以轻松地开发出支持超高并发的 Web 应用和动态网关，正在被越来越多的个人和公司学习和使用。

OpenResty 使用的工作语言是 Lua，它小巧轻便，专为嵌入其他语言而设计，与 C/C++ 具有良好的互操作性，学习成本低，很容易上手。但小并不意味着劣化，与 Python、Ruby 等同级别的脚本语言相比，Lua 在功能上毫不逊色，表 (table) 结构可实现数组、字典、名字空间等诸多特性，而且还有闭包、协程等更灵活更高级的特性。

Lua 语言本身的运行效率就很高，而 OpenResty 为了追求性能的极致，使用的是更高效的 LuaJIT。它利用了汇编语言和即时编译技术，可以把 Lua 脚本程序即时编译成本地机器码，成倍地提升运行速度。LuaJIT 还扩展了 Lua 5.1，加入了 goto 语句和 bit、ffi 等库，进一步增强了 Lua 语言的能力。

OpenResty 的核心是 ngx_lua 模块，它是 Nginx 和 Lua 这两者之间沟通的桥梁。ngx_lua 模块为开发者提供了大量的功能指令和接口调用，让我们能够使用简单的 Lua 语言操纵 Nginx 框架里复杂的流程，自如地穿梭在 init、access、rewrite、content、filter、log 等阶段，处理日志、正则、变量、请求/子请求、定时器、共享内存等 Nginx 内部构件，拼装组合完成任意的业务逻辑。

cosocket 是 OpenResty 中最有价值的功能之一，它基于 Nginx 的连接池机制，能够以直观的同步方式编写 100% 无阻塞的网络通信代码，高效地访问后端的 TCP/UDP 服务，很多 resty 库都是基于它实现的。有了 cosocket，Nginx 的 upstream+load-balance 机制就不是那么重要了。

目前 OpenResty 主要聚焦在 HTTP 协议处理，但 agentzh 也开发了支持 TCP/UDP 协议的 stream_lua 模块。虽然 stream_lua 还处于“实验”阶段，但功能方面已经比较完备，而且大部分 resty 库也可以不加修改地在 stream_lua 环境里运行，值得我们去尝试。

OpenResty 里还有很多有用的功能本章未能详述，例如 coroutine、ssl、thread、config、worker 等，强烈建议读者阅读官方文档继续学习研究。

第 20 章

结束语

本章是全书的结束语，简略讨论本书未能涉及的内容，为读者进一步学习 Nginx 和 OpenResty 开发给出了建议。

20.1 本书的遗憾

Nginx 系统非常复杂，关于它的开发和运维都是很大的话题。由于选题、篇幅、时间和作者水平等因素的限制，本书的内容只是沧海一粟，在广度和深度上做了些不得已的取舍。

虽然本书名为“完全开发指南”，但其实并没有覆盖 Nginx 开发的所有领域，还可以有更多内容奉献出来共同探讨。遗憾的是写作过程中有的文字写出来后个人又觉得不太满意，最终成书时不得不删除，希望以后能有机会再次整理与读者见面。

本书重点讲解了 Nginx HTTP 处理流程中与实际开发相关的要点，同时论述了进程模型、事件驱动、多线程、流处理等其他部分，限于篇幅略去了很多次要的功能，对 Nginx 源代码的分析没有达到逐行逐句的细致程度，也还有很多内部实现细节暂未能深入研究，例如内存池的工作原理、发送数据的限速、server/location 的查找定位、upstream 访问后端服务等，读者可以在书后多加留意。

世上无完美，本书并不期望成为一本面面俱到的“Nginx 百科全书”，懂得放弃，留下一些遗憾，也可以成为将来前进的动力。

20.2 下一步

所谓“行百里者半九十”，阅读本书仅仅是踏入 Nginx 领域的一个起点，要想真正地掌握

Nginx，接下来还需要读者自己去努力。

下面列出了一些进一步研究 Nginx 的方向供读者参考：

- `configure` 脚本是 Nginx 的重要组成部分，它生成 `objs/nginx_modules.c`，决定了 Nginx 使用的模块和顺序；
- 使用磁盘文件，让 Nginx 不局限于使用内存，可以读写磁盘里的数据作为缓冲；
- Nginx 官方源码包含了大量的功能模块，非常有学习价值，阅读它们的源码能够加深理解 Nginx 的工作原理，也可以学习到更多的开发技巧；
- 网络上还存在很多第三方 Nginx 模块和 OpenResty 库，实现了更多更有用的功能，把它们集成进 Nginx 能够大大增强 Nginx 的服务能力；
- 使用 `gdb` 调试是学习 Nginx 的一个有效手段，通过 `gdb` 跟踪到 Nginx 的内部，可以细致地观察它的状态和工作流程；
- 学习 C++11/14 的新特性和 Boost 程序库，了解 C++ 最新最前沿的技术；
- 自己动手编写模块，可以从最简单的小功能开始，开发出可用的模块，勤记笔记，通过实践来逐渐积累 Nginx 的开发经验。

20.3 临别赠言

Nginx 是一个伟大的 Web 服务器，本书在钻研这个伟大作品的方向上做出了一点尝试，愿读者能够在书中汲取力量，在 Web 服务器的开发道路上有所成就。

附录 A

推荐书目

这里列出了一些与本书相关的技术书籍，希望能够帮助读者更好地学习 Nginx 开发。

- [1] Erich Gamma 等著，李英军等译，《设计模式 可复用面向对象软件的基础》。

软件开发历史上里程碑式的著作，设计模式的开山作品，里面提出的 23 个设计模式已经成为软件界的经典，被无数其他论文或书刊引用，也被无数的软件系统所验证并使用。本书是每一个精益求精的程序员都必须拥有的宝典和圣经，可以说是字字珠玑，值得经常翻阅以获取设计灵感。

- [2] Nicolai M. Josuttis 著，侯捷/孟岩译，《C++标准程序库 自修教程与参考手册》。全面分析讲解 C++标准库，1100 余页的煌煌巨著，内容详细丰富，是学习现代 C++语言和标准库的经典书籍，也可能是最好的书籍。

- [3] 罗剑锋著，《Boost 程序库完全开发指南——深入 C++“准”标准库》。

本书是国内较早介绍 Boost 程序库的技术书籍，内容涵盖 Boost 程序库里的所有组件，例如时间与日期、内存管理、容器、算法、操作系统相关、并发编程等，并且使用了最新的 C++11/14 标准，想要了解 C++的最前沿技术不容错过。

- [4] 罗剑锋著，《C++11/14 高级编程——Boost 程序库探秘》。

书目[3]的姊妹篇，深入探讨了 C++11/14 和 Boost 程序库里的迭代器、函数对象、指针容器、侵入式容器、流处理等许多高级组件，还完整地阐述了 C++模板元编程和预处理元编程。

- [5] Stanley B Lippman 等著，王刚等译，《C++ Primer》。

C++入门的经典教材，全面论述了 C++11 标准的方方面面，不仅适合初学者，对于 C++熟手也有很大的参考意义。

- [6] W.Richard Stevens 等著. 尤晋元等译. 《UNIX 环境高级编程》.
UNIX 编程名著, UNIX/Linux 程序员必备, 无须过多介绍。
- [7] W.Richard Stevens 等著. 杨继张译. 《UNIX 网络编程 第 1 卷: 套接口 API》.
UNIX 网络编程的权威著作, 深入介绍 UNIX 下网络编程的诸多细节和注意事项。
- [8] W.Richard Stevens 著. 杨继张译. 《UNIX 网络编程 第 2 卷: 进程间通信》.
接续第 1 卷, 详细介绍了 UNIX 各种进程间的通信机制, 如管道、消息队列、锁、信号量、共享内存等。
- [9] Douglas 等著. 於春景译. 《C++网络编程 卷 1 运用 ACE 和模式消除复杂性》.
ACE 开发者编写的网络中间件著作, 站在领域分析的高度, 详细分析了 C++网络编程相关的若干基本问题, 以及 ACE 是如何解决这些问题的。
- [10] Douglas 等著. 马维达译. 《C++网络编程 卷 2 基于 ACE 和框架的系统化复用》.
本书在卷 1 基础上深入阐述了 ACE 框架的设计原理和诸多用于网络通信的设计模式。

附录 B

GDB调试简介

gdb 是 UNIX/Linux 系统里老牌的调试工具，是研究程序、解决问题的“屠龙刀”。本文简要介绍 gdb 调试 Nginx 的一些基本知识。

设置 Nginx

要使用 gdb 调试 Nginx，需要在 configure 时使用“--with-cc-opt”为 gcc 增加额外的选项，启用编译选项“-g”，并且使用“-O0”禁止编译器优化，即：^①

```
./configure --with-cc-opt="-g -O0" #增加 gcc 的调试选项
```

这样 make 出的 Nginx 程序就带有了调试信息，具备了使用 gdb 调试的基础。

我们还可以在配置文件里使用指令“master_process off”和“daemon off”关闭 Nginx 的 master/worker 进程机制，只在前台运行一个 worker 进程，避免 fork 进程带来的麻烦，简化后续的调试工作。

运行 gdb

可以使用 gdb 直接启动 Nginx，这将使用默认的配置：^②

```
gdb /usr/local/nginx/sbin/nginx #开始调试 Nginx
```

有的时候需要指定配置文件来启动 Nginx，那么就要使用 gdb 的“-args”选项，给出完整的 Nginx 启动参数：

```
gdb --args ./sbin/nginx -c x.conf #指定配置文件调试 Nginx
```

① 也可以使用“CFLAGS="-g -O0" ./configure ...”的方式。

② 运行 Nginx 需要 root 权限，所以 gdb 调试 Nginx 同样也需要 root 权限，或者使用 sudo 命令。

如果 Nginx 已经运行了多个进程, gdb 的 “-p” 选项可以指定要调试的进程号, 直接调试正在运行的 worker 进程:

```
gdb -p xxxx #调试某个正在运行的进程
```

当 Nginx 发生意外导致 core dump 时, 可以指定 core 文件进行调试:

```
gdb /usr/local/nginx/sbin/nginx core-file #用 core 调试 Nginx
```

调试命令

gdb 提供了丰富的调试命令, 可能很多读者都已经很熟悉了, 下面仅列出作者在实际工作中比较常用的一些命令, 供读者参考备忘:

- 回车键 : 重复上一条命令;
- h : 查看帮助信息;
- r : 运行程序启动调试;
- p : 查看变量值;
- watch : 监视变量;
- what : 查看变量的类型;
- pt : 查看变量的真实类型, 非 typedef;
- l : 列出源码;
- b : 设置断点, 可以用函数名或者文件加行号的形式;
- i b : 查看所有的断点信息;
- d : 删除断点;
- c : 继续运行程序;
- n : 步进执行程序, 不会进入函数内部;
- s : 步进执行程序, 会进入函数内部;
- fin : 退出当前函数;
- up : 进入上级函数调用栈;
- down : 进入下级函数调用栈;
- bt : 查看当前函数调用栈信息;
- wh : 启动可视化调试模式, 可以非常直观地看到源码, 离开窗口使用 ^x, ^a;
- q : 退出 gdb。

Nginx C++模块简介

本文简要介绍书内实现的 Nginx C++工具模块 `ngx_cpp_module`，它能够辅助程序员以 C++快速开发 Nginx 模块，GitHub 上的地址是：

https://github.com/chronolaw/nginx_cpp_dev

使用方法

`ngx_cpp_module` 包含了本书内开发的所有 C++类，均是以头文件的形式提供，不需要编译，除 C++11 和 Boost 程序库外没有其他的外部依赖。

`ngx_cpp_module` 也依据 Nginx 开发规范实现了自己的 `config` 脚本，但它没有任何编译动作，只是添加了包含路径支持：

```
#ngx_cpp_module 的 config 脚本
ngx_addon_name=ngx_cpp_module           #模块的名字
HTTP_INCS="$HTTP_INCS $ngx_addon_dir"    #添加包含路径，无编译源码
```

只需在 `configure` 时使用 “`--add-module`” 添加 `ngx_cpp_module`，然后其他模块就可以使用它提供的工具类，非常方便，例如：

```
./configure --add-module=path/to/nginxpp \  #加入 C++工具模块包含路径
            --add-module=xxxx              #其他 C++模块
```

类列表

`ngx_cpp_module` 提供总括性的头文件 `NgxAll.hpp` 和 `NgxStreamAll.hpp`，里面包含了所有的 C++类实现，开发 `http` 模块可以 `include` 前者，而 `stream` 模块需要使用后者。

这些 C++类大都只含有一个指针成员，拷贝的代价极低，非常的轻量级，所以无须实现转移构造函数，默认的拷贝构造函数就足够了。

下面依照出现的章节顺序列出了本书实现的一些 C++ 类，供读者开发时快速参考：

- `NgxUnsetValue` : 泛型的无效值“-1”，简化形式是 `ngx_nil`；
- `NgxValue` : 整数的初始化、合并、UNSET 操作；
- `NgxException` : 错误码的 C++ 异常形式；
- `NgxPool` : 内存池；
- `NgxString` : 引用形式的字符串；
- `NgxClock` : 计时器；
- `NgxDatetime` : 日历日期；
- `NgxLog` : 记录运行日志；
- `NgxArray` : 泛型的动态数组；
- `NgxList` : 泛型的单向链表；
- `NgxQueue` : 泛型的双端队列；
- `NgxRbtree` : 泛型的红黑树；
- `NgxBuf` : 数据缓冲区；
- `NgxChain` : 数据块链表；
- `NgxModuleConfig` : 获取模块的配置信息；
- `NgxModule` : 代理模块实例，获取配置和环境数据；
- `NgxTake` : 简化指令的参数数量设置；
- `NgxModuleCtx` : 操作模块请求相关的环境数据，暂存处理的中间结果；
- `NgxHttpCoreModule`: 向 Nginx 框架注册 http 模块的处理函数；
- `NgxFilter` : 操作 HTTP 过滤链表；
- `NgxRequest` : 处理 HTTP 请求；
- `NgxResponse` : 处理 HTTP 响应；
- `NgxUpstreamHelper`: 转发 HTTP 请求；
- `NgxLoadBalance` : 实现负载均衡算法；
- `NgxVariables` : 使用模板元数据添加 Nginx 变量；
- `NgxVarManager` : 操作 Nginx 变量；
- `NgxComplexValue` : 操作 Nginx 复杂变量，也就是“脚本”；
- `NgxDigest` : 摘要算法，需配合元数据使用。

附录 D

Nginx的字符串格式化

`ngx_sprintf()` 是 Nginx 专门为格式化字符串提供的函数，它模仿了标准 C 函数 `printf()` 的接口和用法，并增加了一些 Nginx 特有的格式化标志，本文列出了这些格式供参考：

- `%u` : 修饰格式，表示无符号整数，如 `size_t`、`u_int` 等；
- `%x` : 修饰格式，以十六进制表示整数；
- `%X` : 修饰格式，功能同 `%x`，但使用大写字母；
- `%T` : 打印 `time_t` 类型的变量；
- `%O` : 打印 `off_t` 类型的变量，可以使用 `%x` 修饰；
- `%z` : 打印 `ssize_t` 或 `size_t` 类型的变量，可以使用 `%u%x` 修饰；
- `%d` : 打印 `int/u_int` 类型的变量，可以使用 `%u%x` 修饰；
- `%l` : 打印 `long` 类型的变量，可以使用 `%u%x` 修饰；
- `%i` : 打印 `ngx_int_t/ngx_uint_t` 类型的变量，可以使用 `%u%x` 修饰；
- `%D` : 打印 `int32_t/uint32_t` 类型的变量，可以使用 `%u%x` 修饰；
- `%L` : 打印 `int64_t/uint64_t` 类型的变量，可以使用 `%u%x` 修饰；
- `%A` : 打印 `ngx_atomic_int_t/ngx_atomic_uint_t`，可以使用 `%u%x` 修饰；
- `%f` : 打印 `double` 类型的变量，可以用 `%.N` 指定小数点后的位数；
- `%P` : 打印 `ngx_pid_t` 类型的变量；
- `%M` : 打印 `ngx_msec_t` 类型的变量；
- `%r` : 打印 `rlim_t` 类型的变量；
- `%p` : 打印指针地址；
- `%V` : 打印 `ngx_str_t*` 类型的变量，注意变量必须是地址形式；
- `%v` : 打印 `ngx_variable_value_t*` 类型的变量，注意变量必须是地址形式；

- %s : 打印标准的 NULL 结尾 C 字符串;
- %*s : 打印指定长度的字符串;
- %Z : 打印 '\0';
- %N : 打印 '\n';
- %c : 打印一个字符;
- %% : 打印一个 % 字符。

下面的代码简单示范了部分格式化标志的用法:

```
ngx_int_t      i    = -100;
unsigned long   ul   = 65535L;
double         f    = 0.618;
ngx_str_t      str  = ngx_string("metroid");
```

```
NgxLogError(r).print(
    "%i,%uL,%.5f,%V,%uxz,%p,%P",
    i, ul, f, &str, ul, str.data, ngx_getpid()
);
```

在 error.log 里的输出是:

```
-100,65535,0.61800,metroid,ffff,0000000000482212,8686
```

附录 E

nginxScript简介

2015 年, Nginx 正式宣布支持 JavaScript, 允许在配置文件里使用专用指令嵌入 JavaScript 代码, 从而能够更加轻松地动态化配置 Nginx——这同时也方便了那些前后端“通吃”的全栈工程师, 使他们可以比较容易地从 node.js 这样的服务器框架迁移到更优秀更高效的 Nginx。

Nginx 内置的 JavaScript 被称为 nginxScript, 是 ECMAScript 5.1 的一个精简改造版, 没有闭包、内建对象、垃圾回收等特性, 利于学习使用。

nginxScript 使用的虚拟机不是流行的 V8、SpiderMonkey、JavaScriptCore 等引擎, 完全是由 Nginx 开发团队自行实现的, 适应 Nginx 的运行环境, 为每个请求创建一个独立的超轻量级 VM, 保证了效率和安全。

经过了近三年的发展, nginxScript 终于走出了“experimental”阶段, 标记为“available and stable”状态, 但目前功能仍不够完善, 只能做一些较简单的配置管理和记录日志的工作, 未来还有很多的发展空间。

编译集成

nginxScript 由 njs 模块实现, 它不包含在 Nginx 源码里, 需要我们自行下载编译集成, 例如:

<pre>sudo apt-get install mercurial</pre>	#Nginx 使用 Mercurial 管理源码
<pre>hg clone http://hg.nginx.org/njs</pre>	#可以用类似 git 的方式下载源码
<pre>./configure --add-module=path/to/njs/nginx</pre>	#静态链接方式集成进 Nginx

功能指令

nginxScript 在 HTTP 框架和 Stream 框架里都可以工作, 但支持的指令略有差异。

nginxScript 在 `http{}` 里可用的指令有：^①

- `js_include` : 指定使用的 nginxScript 脚本文件;
- `js_set` : 类似 `set` 指令, 设置变量值;
- `js_content` : 在 `content` 阶段处理请求, 产生响应内容。

nginxScript 在 `stream{}` 里可用的指令有:

- `js_include` : 指定使用的 nginxScript 脚本文件;
- `js_set` : 类似 `set` 指令, 设置变量值;
- `js_access` : 在 `access` 阶段处理请求, 执行访问控制功能;
- `js_preread` : 在 `preread` 阶段处理请求, 检查数据;
- `js_filter` : 在 `filter` 阶段处理请求, 过滤数据。

使用 nginxScript 前必须编写一个 nginxScript 脚本文件, 里面是 `http{}` 或 `stream{}` 里用到的所有函数, 然后用指令 `js_include` 指定, 它只能出现一次——也就是说, 一个 Nginx 实例最多只能有两个 nginxScript 脚本文件, `http` 和 `stream` 各一个。

指定 nginxScript 脚本文件可以用绝对路径或相对路径, 相对路径的起始位置是 Nginx 的配置目录 (即 `/conf`)。

代码示例

Nginx 官网的文档较详细地介绍了 `njs` 模块的接口和用法, 限于篇幅这里不再重复。

下面的代码实现了两个简单的 nginxScript 函数:

```
function hello(req, res)                                //函数必须有 req 和 res 两个参数
{
    req.log("hello nginxScript");                        //记录日志
    return "hello nginxScript";                          //返回一个字符串
}

function reply(req, res)                                //函数必须有 req 和 res 两个参数
{
    var str;

    str = "js content: ";                                //准备字符串
    str += req.method + ",";                             //获取请求方法
    str += req.uri + ",";                                //获取 uri
}
```

① 早期版本的 nginxScript 提供 `js_set` 和 `js_run` 两个指令, 与现在的版本不同。

```
str += req.variables.hello + ".\n";           //获取变量

res.contentLength = str.length;               //设置响应数据的长度
res.status = 200;                             //设置响应码

res.sendHeader();                             //发送响应头
res.send(str);                                //发送响应体
res.finish();                                 //结束处理请求
}
```

然后需要在配置文件的 `http{}` 里使用 `js_include` 设置脚本文件和变量（注意不能在 `server{}` 或 `location{}` 里）:

```
js_include    js/http.js;                    #脚本存放在 conf/js 里
js_set        $hello hello;                  #调用函数 hello() 设置变量
```

在 `server{}` 里定义访问的 `location`:

```
location /js {
    js_content reply;                         #调用函数 reply() 产生内容
}
```

使用 `curl` 访问, 结果如下:

```
curl 127.0.0.1/js -v                          #调用 curl 访问服务
js content:GET,/js,hello nginxScript           #输出 nginxScript 产生的结果
```

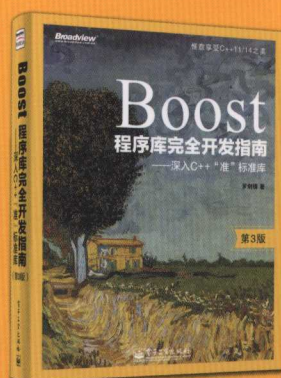
读者服务

轻松注册成为博文视点社区用户 (www.broadview.com.cn), 扫码直达本书页面。

- **提交勘误:** 您对书中内容的修改意见可在【提交勘误】处提交, 若被采纳, 将获赠博文视点社区积分 (在您购买电子书时, 积分可用来抵扣相应金额)。
- **与作者交流:** 在页面下方【读者评论】处留下您的疑问或观点, 与作者和其他读者一同学习交流。

页面入口: <http://www.broadview.com.cn/31457>





Boost程序库完全开发指南
深入C++“准”标准库（第3版）
ISBN 978-7-121-25313-3

拒绝堆砌臃肿,支持纯正原创

出版事宜请关注  新浪微博 @ 半亩方塘 _
weibo.com

投稿邮箱: sxy@phei.com.cn

加入本书读者 QQ 群 465398813,以书会友,资源共享!

Nginx 完全开发指南

使用C、C++和OpenResty

我很感谢罗剑锋能在他的这本新书中帮忙推广和普及OpenResty这个开源Web平台的相关技术。我们最初做OpenResty和ngx_lua等模块的目的其实就是希望开发者能尽量少写Nginx的C/C++模块，多写Lua以及我们即将推出的像fanlang和edgelang这样更上层的高级编程语言。毕竟我们自己也深知Nginx级别的C/C++编程的不易。

无论如何，能有像罗剑锋这样的热心开发者，把Nginx C/C++编程的主要方面比较详细地写下来，也是非常有意义的事情。

章亦春

OpenResty 开源平台的作者，OpenResty Inc. 公司首席执行官



博文视点Broadview



@博文视点Broadview



策划编辑：孙学瑛
责任编辑：安娜
封面设计：侯士卿

上架建议：Web服务器 > Nginx

ISBN 978-7-121-31457-5



9 787121 314575 >

定价：99.00元